

amira[®] 4.1

amira User's Guide

Copyright Information

©1995-2005 Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB), Germany

©1999-2005 Mercury Computer Systems.

All rights reserved.

Trademark Information:

amira is being jointly developed by Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) and Mercury Computer Systems.

amira ® is a registered trademark of Konrad-Zuse-Zentrum für Informationstechnik Berlin.

HardCopy, MeshViz, VolumeViz, TerrainViz are trademarks of Mercury Computer Systems S.A.

Mercury Computer Systems S.A. is a source licensee of OpenGL®, Open Inventor® from Silicon Graphics, Inc. OpenGL® and Open Inventor® are registered trademarks of Silicon Graphics, Inc.

All other products and company names are trademarks or registered trademarks of their respective companies.

This manual has been prepared for Mercury Computer Systems licensees solely for use in connection with software supplied by Mercury Computer Systems and is furnished under a written license agreement. This material may not be used, reproduced or disclosed, in whole or in part, except as permitted in the license agreement or by prior written authorization of Mercury Computer Systems. Users are cautioned that Mercury Computer Systems reserves the right to make changes without notice to the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including but not limited to typographical, arithmetic or listing errors.

Contents

I	amira User's Guide	1
1	Introduction	3
1.1	Overview	3
1.2	Features	4
1.2.1	Direct Volume Rendering	5
1.2.2	Isosurfaces	5
1.2.3	Segmentation	5
1.2.4	Surface Reconstruction	5
1.2.5	Surface Simplification	6
1.2.6	Generation of Tetrahedral Grids (Mesh Pack)	6
1.3	Application Areas	6
1.4	Packs	7
2	First steps in amira	9
2.1	Getting Started	10
2.1.1	Loading Data	10
2.1.2	Invoking Editors	13
2.1.3	Visualizing Data	13
2.1.4	Interaction with the Viewer	14
2.2	How to load image data	16
2.2.1	The amira file browser	17
2.2.2	Reading 3D image data from multiple 2D slices	17

2.2.3	Setting the bounding box	19
2.2.4	The Stacked Slices file format	19
2.2.5	Working with Large Disk Data	20
2.3	Visualizing 3D Images	22
2.3.1	Orthogonal Slices	23
2.3.2	Simple Data Analysis	23
2.3.3	Resampling the Data	24
2.3.4	Displaying an Isosurface	25
2.3.5	Cropping the Data	26
2.3.6	Volume Rendering	26
2.4	Segmentation of 3D Images	28
2.4.1	Interactive Image Segmentation	29
2.4.2	Volume Measurement	30
2.4.3	Threshold Segmentation	31
2.4.4	Refining Threshold Segmentation Results	31
2.5	Surface Reconstruction from 3D Images	33
2.5.1	Extracting Surfaces from Segmentation Results	33
2.5.2	Simplifying the Surface	33
2.6	Creating a Tetrahedral Grid from a Triangular Surface (Mesh Pack)	34
2.6.1	Simplifying the Surface	35
2.6.2	Editing the Surface	36
2.6.3	Generation of a Tetrahedral Grid	38
2.7	Warping and Registration Using Landmarks	40
2.7.1	Displaying Data Sets in Two Viewers	40
2.7.2	Creating a Landmark Set	40
2.7.3	Registration via a Rigid Transformation	43
2.7.4	Warping Two Image Volumes	43
2.8	Registration of 3D image datasets	43
2.8.1	Basic Manual Registration	44
2.8.2	Automatic Registration	45

2.8.3	Image Fusion	46
2.9	Alignment of 2D Physical Cross-subsections	47
2.9.1	Basic Manual Alignment	48
2.9.2	Alignment Via Landmarks	49
2.9.3	Optimizing the Quality Function	51
2.9.4	Resampling the Input Data	52
2.9.5	Using a Reference Image	52
2.10	Visualization of Vector Fields (Mesh Pack)	53
2.10.1	Loading the Wing and the Flow Field	53
2.10.2	Line Integral Convolution	54
2.10.3	Illuminated Stream Lines	55
2.11	Creating animated demonstrations	57
2.11.1	Creating a Network	57
2.11.2	Animating an OrthoSlice module	58
2.11.3	Activating a module in the viewer window	60
2.11.4	Using a camera rotation	62
2.11.5	Editing or removing an already defined event	62
2.11.6	Overlaying the bone with skin	63
2.11.7	Using clipping to add the skin gradually	64
2.11.8	More comments on clipping	66
2.11.9	Breaks and Function Keys	66
2.11.10	Loops and go-to	68
2.11.11	Storing and replaying the animation sequence	68
2.12	Creating movie files	69
2.12.1	Attaching MovieMaker to a camera path	69
2.12.2	Attaching MovieMaker to DemoMaker	71
2.13	Using MATLAB Scripts	72
2.13.1	Lowpass filtering on images	72
2.13.2	Thresholding on a volume	73

3 Program Description 77

3.1	Interface Components	77
3.1.1	File Menu	77
3.1.2	Edit Menu	80
3.1.3	Pool Menu	81
3.1.4	Create Menu	83
3.1.5	View Menu	83
3.1.6	Online Help	85
3.1.7	Main Window	88
3.1.8	Viewer Window	92
3.1.9	Console Window	95
3.1.10	File Dialog	97
3.1.11	Job Dialog	99
3.1.12	Preferences Dialog	100
3.1.13	Snapshot Dialog	104
3.1.14	System Information Dialog	105
3.2	General Concepts	105
3.2.1	Class Structure	106
3.2.2	Scalar Field and Vector Fields	107
3.2.3	Coordinates and Grids	108
3.2.4	Surface Data	109
3.2.5	Vertex Set	109
3.2.6	Transformations	109
3.2.7	Parameters	110
4	Technical Information	111
4.1	Data Import	111
4.2	Command Line Options	112
4.3	Environment Variables	113
4.4	User-defined start-up script	114
4.5	System Requirements	115
4.5.1	On System Stability	116

4.5.2	Microsoft Windows	116
4.5.3	Silicon Graphics	116
4.5.4	HP-UX	117
4.5.5	SunOS	117
4.5.6	Linux	117
4.5.7	Mac	118
4.6	amira and the /3GB switch	118
5	Scripting	121
5.1	Introduction	121
5.2	Introduction to Tcl	122
5.2.1	Tcl Lists, Commands, Comments	122
5.2.2	Tcl Variables	123
5.2.3	Tcl Command Substitution	124
5.2.4	Tcl Control Structures	124
5.3	amira Script Interface	128
5.3.1	Predefined Variables	129
5.3.2	Object commands	130
5.3.3	Global commands	130
5.4	amira Script File	141
5.5	Configuring Popup Menus	142
5.6	Registering pick callbacks	145
II	Molecular Pack User's Guide	147
6	Molecular Pack Introduction	149
6.1	First Steps with Molecular Visualization in amira	149
6.1.1	Getting Started with Molecular Visualization	150
6.1.2	Selection, Labeling, and Masking	152
6.1.3	Alignment of Molecules	159
6.1.4	Molecular Surfaces	163

6.1.5	Sequential and Structural Alignment	166
6.1.6	Editing of molecules	167
6.1.7	Molecular Interfaces	170
6.1.8	Measurement	172
6.2	Molecular Data Structures	173
6.2.1	Internal Structure of Molecules	174
6.3	Displaying Molecules	174
6.3.1	Coloring Molecules	174
6.3.2	Selecting and Filtering atoms	176
6.4	Aligning Molecules	177
6.4.1	Alignment of Trajectories	177
6.4.2	Mean Distance Alignment	179
6.4.3	Sequence alignment	179
6.5	Visualizing Molecular Trajectories and Metastable Conformations	180
6.6	Atom Expressions	180
6.6.1	Overview	180
6.6.2	Grammar	181
6.6.3	Literals	181
6.6.4	Operators	182
6.6.5	Shortcuts	183
6.6.6	Further Examples	183

III VR Pack User's Guide 185

7 VR Pack Configuration 187

7.1	VR Pack essentials	188
7.2	Flat screen configurations	188
7.2.1	Example: A two-channel passive stereo configuration	189
7.2.2	Example: A super-wide configuration with soft-edge blending	191
7.2.3	Example: A tiled four-channel 2x2 monitor configuration	193

7.3	Immersive configurations	194
7.3.1	Example: A Workbench configuration	195
7.3.2	Example: A Holobench configuration	197
7.3.3	Example: A 4-side CAVE configuration	199
7.4	Calibrating the tracking system	201
7.5	The VR Pack cluster version	203
7.6	The VR Pack service	205
8	VR Pack Interaction	209
8.1	3D user interaction	209
8.1.1	The 3D menu	210
8.1.2	Tracker Emulator	210
8.2	User-defined 3D menu items	212
8.2.1	3D module controls	215
8.2.2	Navigation modes	216
8.2.3	Tcl event procedures	216
8.2.4	The 2D mouse mode	217
8.3	Writing VR Pack custom modules	218
IV	Large Data Pack User's Guide	223
9	Large Data Pack User's Guide	225
9.1	Working with out-of-core data files (LDA)	225
9.1.1	Tune the size threshold to allow conversion	225
9.1.2	Load the out-of-core data	226
9.1.3	Raw data parameters	227
9.1.4	Out-of-core conversion	227
9.1.5	Display an ortho slice, an oblique slice, and a 3D volume	227
V	Quantification+ Pack User's Guide	233

Part I

amira User's Guide

Chapter 1

Introduction

amira is a 3D visualization and modelling system. It allows you to visualize scientific data sets from various application areas, e.g. medicine, biology, chemistry, physics, or engineering. 3D objects can be represented as grids suitable for numerical simulations, notably as triangular surface and volumetric tetrahedral grids. **amira** provides methods to generate such grids from voxel data representing an image volume, and it includes a general purpose interactive 3D viewer. Section 1.1 (Overview) provides a short overview of the fundamentals of **amira**, i.e. its object-oriented design and the concept of data objects and modules.

Section 1.2 (Features) summarizes key features of **amira**, for example direct volume rendering, image processing, and surface simplification.

Section 1.3 (Application Areas) illustrates some typical application areas of **amira** and shows what kinds of problems can be solved using the system.

Section 1.4 (Packs) shortly describes optional packs available for **amira** and what they can be used for.

1.1 Overview

amira is a modular and object-oriented software system. Its basic system components are modules and data objects. Modules are used to visualize data objects or to perform some computational operations on them. The components are represented by little icons in the *Pool*. Icons are connected by lines indicating processing dependencies between the components, i.e., which modules are to be applied to which data objects. Data objects of specific types are created automatically from file input data when reading such or as output of module computations, modules matching an existing data object are created as instances of particular module types via a context-sensitive popup menu. Networks can be created with a minimal amount of user interaction. Parameters of data objects and modules can be modified in **amira**'s interaction area.

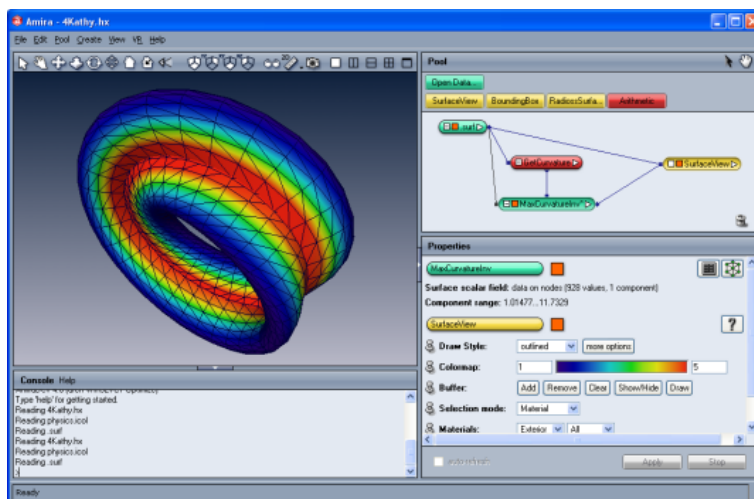


Figure 1.1: Data objects and modules are represented as little icons in the Pool (top right). In the 3D graphic window a surface colored according to its curvature is shown. Curvature information has been computed by a computational module and is stored as a separate data object. In the mid right window the parameters of selected modules are shown. The lower left pane provides a Tcl-command shell as well as access to the help browser.

For some data objects such as surfaces or colormaps there exist special-purpose interactive editors that allow the user to modify the objects. All *amira* components can be controlled via a Tcl command interface. Commands can be read from a script file or issued manually in a separate console window.

The biggest part of the screen is occupied by a 3D graphics window. Additional 3D views can be created if necessary. *amira* is based on the latest release of the Open Inventor from Mercury. In addition, several modules apply direct OpenGL rendering to achieve special rendering effects or to maximize performance. In total, there are more than 120 data object and module types. They allow the system to be used for a broad range of applications. User-defined extensions are facilitated by the *amira* developer version.

1.2 Features

amira provides a large number of module types allowing you to visualize various kinds of scientific data as well as to create polygonal models from 3D images. All visualization techniques can be arbitrarily combined to produce a single scene. Moreover, multiple data sets can be visualized simultaneously, either in several viewer windows or in a common one. A built-in transformation editor makes it easy to register data sets with respect to each other or to deal with different coordinate systems.

1.2.1 Direct Volume Rendering

One of the most intuitive and most powerful techniques for visualizing 3D image data is *direct volume rendering*. Light emission and light absorption parameters are assigned to each point of the volume. Simulating the transmission of light through the volume makes it possible to display your data from any view direction without constructing intermediate polygonal models. By exploiting modern graphics hardware, **amira** is able to perform direct volume rendering almost in realtime, even for data volumes of 40 megabytes and more. Volume rendered images can be combined with any type of polygonal display. This improves the usefulness of this technique significantly. Moreover, multiple data sets can be volume rendered simultaneously – a unique feature of **amira**. Transfer functions with different characteristics required for direct volume rendering can either be generated automatically or edited interactively using an intuitive colormap editor.

1.2.2 Isosurfaces

Isosurfaces are most commonly used for analyzing arbitrary scalar fields sampled on a discrete grid. Applied to 3D images, the method provides a very quick, yet sometimes sufficient method for reconstructing polygonal surface models. Beside standard algorithms, **amira** provides an improved method which generates significantly fewer triangles with very little computational overhead. In this way large 3D data sets can be displayed interactively even on smaller desktop graphics computers. Like other polygonal models, isosurfaces can be colored in order to visualize a second independent data set. Another highlight comprises the realistic view-dependent way of rendering semi-transparent surfaces. By correlating transparency with local orientation of the surface relative to the viewing direction, complex spatial structures can be understood much more easily.

1.2.3 Segmentation

amira also provides a component for 3D image segmentation with several special-purpose features. This component is called image segmentation editor. It offers a large set of segmentation tools, ranging from purely manual to fully automatic. Among others, the following tools are provided: brush (painting), lasso (contouring), magic wand (region growing), thresholding, intelligent scissors, contour fitting (snakes), contour interpolation and extrapolation, various filters including smoothing, cleaning, and connected component analysis. Although the display is slice-oriented, many tools can be applied in both 2D and 3D. Since the editor does not store contours surrounding regions but region labels, a unique and well-defined classification is guaranteed.

1.2.4 Surface Reconstruction

Once the interesting features in a 3D image volume have been segmented, **amira** is able to create a corresponding polygonal surface model. The surface may have non-manifold topology if there are locations where three or more regions join. Even in this case the polygonal surface model is guaranteed to be topologically correct, i.e. free of self-intersections. Fractional weights which are automatically

generated during segmentation allow the system to produce smooth boundary interfaces. This way realistic high-quality models can be obtained, even if the underlying image data are of low resolution or contain severe noise artifacts. Making use of innovative acceleration techniques, surface reconstruction can be performed very quickly. Moreover, the algorithm is robust and fail-safe.

1.2.5 Surface Simplification

Surface simplification is another prominent feature of **amira**. It can be used to reduce the number of triangles in an arbitrary surface model according to a user-defined value. Thus, models of finite-element grids, suitable for being processed on low-end machines, can be generated. The underlying simplification algorithm is one of the most elaborate ones available. It is able to preserve topological correctness, i.e., self-intersections commonly produced by other methods are avoided. In addition, the quality of the resulting mesh, according to measures common in finite element analysis, can be controlled. For example, triangles with long edges or triangles with bad aspect ratio can be suppressed.

1.2.6 Generation of Tetrahedral Grids (Mesh Pack)

Mesh Pack allows you not only to generate surface models from your data but also to create true volumetric tetrahedral grids suitable for advanced 3D finite-element simulations. These grids are constructed using a flexible advancing-front algorithm. Again, special care is taken to obtain meshes of high quality, i.e., tetrahedra with bad aspect ratio are avoided. Several different file formats are supported, so that the grid can be exported to many standard simulation packages. In the developer version additional file formats can easily be added by the user.

1.3 Application Areas

amira is successfully being used in a number of different application areas. Among these are:

- Medicine
- Biology
- Material Sciences
- Computational Fluid Dynamics
- Physics
- Geophysics
- Astrophysics

Examples from these disciplines are illustrated by several *demo scripts* contained in the online version of the user's guide.

1.4 Packs

amira packs are additional sets of modules providing solutions for dedicated application areas. Packs can be added to a standard **amira** installation at any time. For each pack a separate license is required. Currently, the following packs are available for **amira** 4.1:

- **amira Developer Pack** allows you to develop your own custom modules, file readers, and file writers using the C++ programming language.
- **amira VR Pack** is designed to enable the use of **amira**'s advanced data visualization and analysis features on immersive VR systems and tiled screen configurations. It has built-in support for efficient multi-threaded rendering on multipipe systems and for distributed rendering on cluster systems using application-level distribution. This approach leads to optimal performance with minimal bandwidth requirements. Tracking capabilities allow for a true immersive experience as well as interaction with the visualization.
- **amira Molecular Pack** adds advanced tools for the visualization of molecules. It combines **amira**'s strong capabilities for 3D data visualization such as hardware-accelerated volume rendering, with specific tools for molecular visualization and data analysis, such as molecular surfaces, sequence alignment, configuration density computation, and molecule trajectories. **amira Molecular Pack** includes a very powerful molecule editor.
- **amira Mesh Pack** supports mesh generation for flow, stress, and thermal analysis; for export of surface or 3D meshes to solvers; and for post-processing of data coming back from these solvers, providing very powerful visualization on scalar, vector, and tensor fields.
- **amira Large Data Pack** allows **amira** to deal with data sets up to 8GB. This allows you go beyond the limit of 32-bits addressing and average size data sets. For data sets smaller than 1GB, a Large Data Pack license is not necessary.
- **amira Very Large Data Pack** manages and visualizes very large amounts of volume data, up to hundreds of gigabytes. The multi-resolution technique used in this pack allows for interactive visualization and navigation through vast amounts of data.
- **amira Quantification+ Pack** includes new image processing capability as well as image analysis and quantification tools. Very powerful for material sciences, this pack allows for efficient statistical analysis of data in this application area.
- **amira DICOM Reader** allows for import/export of DICOM data in **amira**, making **amira** the perfect application for medical data analysis.
- **amira SEG-Y Reader** allows for import of SEG-Y data in **amira**, making **amira** a very powerful application for building visualizations on oil and gas data sets. Coupled with the **amira Very Large Data Pack**, it allows for exploration of huge reservoir data sets.

- **CATIA 4, CATIA 5, IGES, and STEP readers** for **amira** allow the visualization data coming from the CAD field. Coupled with **amira VR Pack**, these readers bring CAD models to immersive visualization and collaborative environments.
- **TNO Madymo For amira** allows **amira** to build visualizations from data in the Madymo file format. MADYMO is the world-wide standard for occupant safety analysis. It is used extensively in industrial engineering, design offices, research laboratories, and technical universities.
- **Mecalog Radioss For amira** allows **amira** to build visualizations from data in the Radioss file format. Radioss from Mecalog is a CAE software package for realistic industrial simulations of applied mechanics based on explicit finite element techniques.
- **ResolveRT Pack** supports microscopy features as well as deconvolution.
- **Skeleton Pack** supports reconstruction and analysis of vascular and dendritic networks.

The following table shows the license keyword associated with each of the **amira** packs:

amira Pack	License keyword
Developer Pack	AmiraDev
VR Pack	AmiraVR
Molecular Pack	AmiraMol
Mesh Pack	AmiraMesh
Large Data Pack	AmiraLD
Very Large Data Pack	AmiraVLD
Quantification+ Pack	AmiraQuant
DICOM reader	AmiraDicomReader
SEG-Y reader	AmiraSegyReader
CATIA5 reader	AmiraCatia5Reader
CATIA4 reader	AmiraCatia4Reader
IGES reader	AmiraIgesReader
STEP reader	AmiraStepReader
TNO Madymo reader	AmiraMadymoReader
Mecalog Radioss reader	AmiraRadiossReader
ResolveRT Pack	ResolveRT
Skeleton Pack	AmiraSkel

For additional information about **amira** and its packs, please refer to the **amira** web site, www.mc.com/tgs.

Chapter 2

First steps in amira

This chapter contains step-by-step tutorials illustrating the use of **amira**. The tutorials are almost independent of each other, so after reading the basics in the Getting Started section it is possible to follow each tutorial without knowing the others. If you go through all tutorials you will get a good survey of **amira**'s basic features. In particular, these topics will be covered:

- *Getting started* - the basics of **amira**
- *Reading images* - how to read images
- *Visualizing 3D images* - slices, isosurfaces, volume rendering
- *Image segmentation* - segmentation of 3D image data
- *Surface reconstruction* - surface reconstruction from 3D images
- *Grid generation* - creating a tetrahedral grid from a triangular surface
- *Warping* - how to work with landmark sets
- *3D image registration* - how to register 3D image datasets
- *Alignment of 2D Physical Cross-Sections* - how to reconstruct a 3D model
- *Vector fields* - stream lines and other techniques
- *The DemoMaker module* - creating animations with the DemoMaker module
- *Creating movie files* - how to use the MovieMaker module
- *Using MATLAB* - how to use the CalculusMatlab module

In all tutorials the steps to be performed by the user are marked by a dot. If you only want to get a quick idea how to work with **amira** you may skip the explanations between successive steps and just follow the instructions. But in order to get a deeper understanding you should refer to the text.

Note: If you want to visualize your own data, please first refer to Section [4.1](#). This section contains some general hints on how to import data sets into **amira**.

2.1 Getting Started

In this section you will learn how to

1. start the program
2. load a demo data set into the system
3. invoke editors for editing the data
4. connect visualization modules to the data
5. interact with the 3D viewer.

The following text has the form of a short step-by-step tutorial. Each step builds on the steps described before. We recommend that you read the text online and carry out the instructions directly on the computer. Instructions are indicated by a dot so you can execute them quickly without reading the explanations between the instructions.

- On a Windows system, select the **amira** icon from the start menu. On a Unix system, start **amira** by entering `amira` in a shell window.

If there is no such command, the software has not been properly installed. In this case try to execute the script `bin/start` located in the **amira** root directory.

When **amira** is running, a window like the one shown in Figure 2.1 appears on the screen. The user interface is divided into four major regions. The 3D viewer window displays visualization results, e.g., slices or isosurfaces. The Pool will contain small icons representing data objects and modules. The Properties Area displays interface elements (*ports*) associated with **amira** objects. Finally, the lower left pane is shared by the console and **amira**'s integrated hypertext help browser. Click on the Console or Help tab to select which window you want to view. The console prints system messages and lets you enter **amira** commands. You can use the help browser to read the user's guide online.

You can also activate the help browser by pressing F1, selecting *User's Guide* from the *Help* menu of **amira**'s main window, or by typing `help` in the console window.

2.1.1 Loading Data

Usually, the first thing you will do after starting **amira** is to load a data set. Let's see how this can be done:

- Choose *Open Data ...* from the *File* menu.

After selecting this menu item, the file dialog appears (see Figure 2.2). By default the dialog displays the contents of the first directory defined in the environment variable `AMIRA_DATADIR`. If no such variable exists the contents of **amira**'s demo data directory are displayed. You can quickly switch to

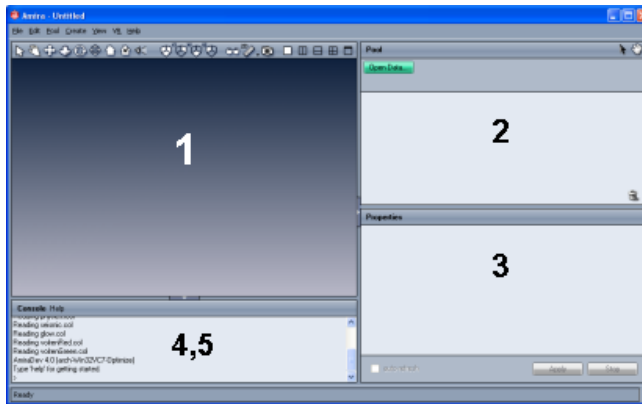


Figure 2.1: The amira user interface consists of five major parts: the 3D viewer (1), the Pool (2), the Properties Area (3), the console window (4), and the help browser (5).

other directories, e.g., to the current working directory, using the directory list located in the upper part of the dialog window.

At the top of the Pool is an Open Data button which is a shortcut to the File/Open Data dialog. You may use it for opening data files in the tutorials that follow. However, the tutorials will instruct you to use the File/Open Data command.

amira is able to determine many file formats automatically, either by analyzing the file header or the file name suffix. The format of a particular file will be printed in the file dialog right beside the file name.

Now, we would like to load a scalar field from one of the demo data directories contained in the amira distribution.

- Change to the directory `data/tutorials`, select the file `lobus.am` and press *OK*.

The data will be loaded into the system. Depending on its size this may take a few seconds. The file is stored in amira's native *AmiraMesh* format. The file `lobus.am` contains 3D image data of a part of a fruit fly's brain, namely an optical lobe, obtained by confocal microscopy. This means the data represents a series of parallel 2D image slices across a 3D volume. Once it has been loaded, the data set appears as a green icon in the Pool. In the following we call this data set "lobus data set".

- Click on the green data icon with the left mouse button to select it.

This causes some information about the data record to be displayed in the Properties Area (Figure 2.3). In our case we can read off the dimensions of the data set, the primitive data type, the coordinate type, as well as the voxel size. To deselect the icon, click on an empty area in the Pool window. You may

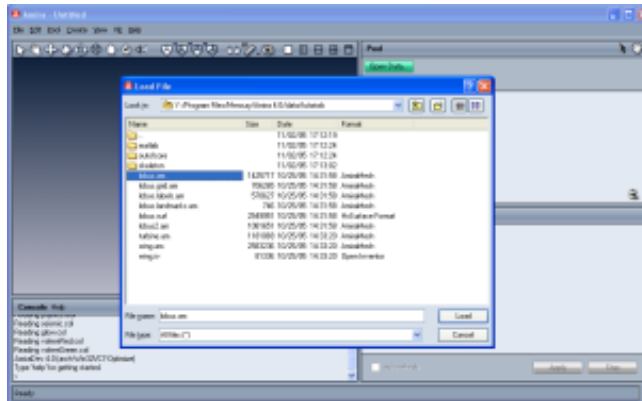


Figure 2.2: Data sets can be loaded into amira using the file browser. In most cases, the file format can be determined automatically. This is done by either analyzing the file header or the file name suffix.

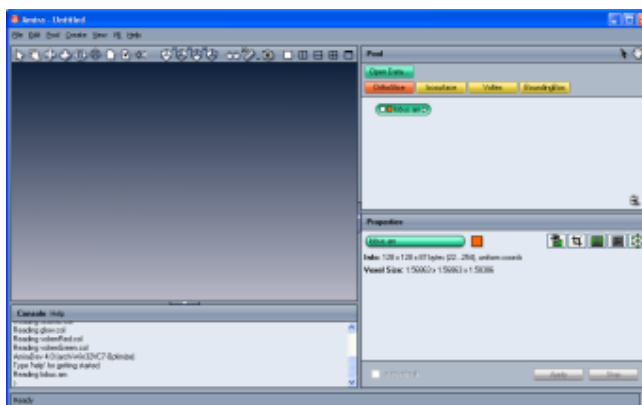


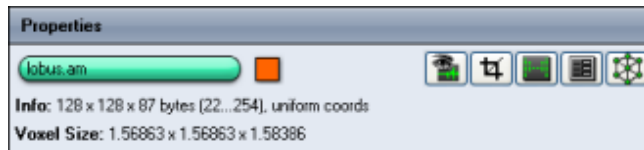
Figure 2.3: Data objects are represented by green icons in the Pool. Once an icon has been selected information about the data set such as its size or its coordinate type is displayed in the Properties Area.

also pick the icon with the left mouse button and drag it around in the Pool.

Clicking on an object typically causes additional buttons to be displayed in the button area at the top of the Pool. These buttons are convenience buttons allowing easy one-click access to the modules most frequently used by the selected object. The tutorials, however, will have you to access modules via the menu interface to help familiarize you with the organization of modules within *amira*.

2.1.2 Invoking Editors

After selecting an object, in addition to the textual information, some buttons appear in the Properties Area, to the far right of the data object's name. These buttons represent *editors* which can be used to interactively manipulate the data object in some way. For example, all data objects provide a *parameter editor*. This editor can be used to edit arbitrary attributes associated with the data set, e.g. filename, original size, or bounding box. Another example is the *transform editor* which can be used to translate or rotate the data in world coordinates. However, at this point we don't want to go into details. We just want to learn how to create and delete an editor:



- Invoke one of the editors by clicking on an editor icon.
- Close the editor by clicking again on the editor icon.

Further information about particular editors is provided in the user's reference manual.

2.1.3 Visualizing Data

Data objects like the lobus data can be visualized by attaching *display modules* to them. Each icon in the Pool provides a popup menu from which matching modules, i.e., modules that can operate on this specific kind of data, can be selected. To activate the popup menu

- click with the right mouse button on the green data icon. Choose the entry called *BoundingBox*.

After you release the mouse button, a new *BoundingBox* module is created and is automatically connected to the data object. The *BoundingBox* object is represented by a yellow icon in the Pool and the connection is indicated by a blue line connecting the icons. At the same time, the graphics output generated by the *BoundingBox* module becomes visible in the 3D viewer. Since the output is not very interesting, in this case we will connect a second display module to the data set:

- Choose the entry called *OrthoSlice* from the popup menu of the lobus data set.

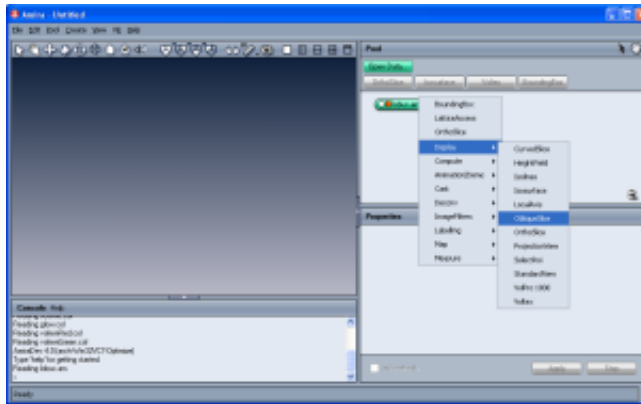


Figure 2.4: In order to attach a module to a data set, click on the green icon using the right mouse button. A popup menu appears containing all modules which can be used to process this particular type of data.

Now a 2D slice through the optical lobe is shown in the viewer window. Initially, a slice oriented perpendicular to the z-direction and centered inside the image volume is displayed. Slices are numbered 0, 1, 2, and so on. The slice number as well as the orientation are parameters of the *OrthoSlice* module. In order to change these parameters, you must select the module. Like for the green data icon, this is done by clicking on the *OrthoSlice* icon with the left mouse button. By the way, in contrast to the *BoundingBox*, the *OrthoSlice* icon is orange, indicating that this module can be used for clipping.

- Select the *OrthoSlice* module.

Now you should see various buttons and sliders in the Properties Area, ordered in rows. Each row represents a *port* allowing you to adjust one particular control parameter. Usually, the name of a port is printed at the beginning of a row. For example, the port labeled *Slice Number* allows you to change the slice number via a slider.

- Select different slices using the *Slice Number* port.

By default, *OrthoSlice* displays slices with axial orientation, i.e., perpendicular to the z-direction. However, the module can also extract slices from the image volume perpendicular to x- and y-direction. These two alternate orientations are sometimes referred to as *sagittal* and *coronal* (these are standard phrases used in radiology).

2.1.4 Interaction with the Viewer

The 3D viewer lets you look at the model from different positions. If you click on the *Trackball* button in the viewer toolbar, moving the mouse inside the viewer window with the left mouse button pressed

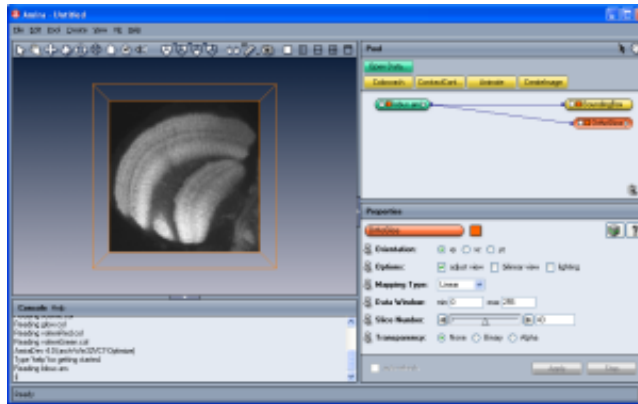


Figure 2.5: Visualization results are displayed in the 3D viewer window. Parameters or ports of a module are displayed in the Properties Area after you select the module.

lets you rotate the object. If you click on the *Translate* or the *Zoom* buttons, you can translate or zoom the object. (For zoom, move the mouse up and down.)

Alternatively, with the middle mouse button pressed you can translate the object. For zooming press both the left and the middle mouse buttons at the same time and move the mouse up or down.

Notice that the mouse cursor has the shape of a little hand inside the viewer window. This indicates that the viewer is in viewing mode. By pressing the `ESC` key you can switch the viewer into interaction mode. In this mode, interaction with the geometry displayed in the viewer is possible by mouse operations. For example, when using *OrthoSlice* you can change the slice number by clicking on the slice and dragging it.

- Select different buttons of the *Orientation* port of the *OrthoSlice* module.
- Rotate the object in a more general position.
- Disable the *adjust view* toggle in the *Options* port.
- Change the orientation using the *Orientation* port again.
- Choose different slices using the *Slice Number* port or directly in the viewer with the interaction mode described above.

Each display module has a *viewer toggle* by which you can switch off the display without removing the module. This button is just to the right of the colored bar where the module name is shown as illustrated below.

- Deactivate and activate the display of the *OrthoSlice* or *BoundingBox* module using the *viewer toggle*.

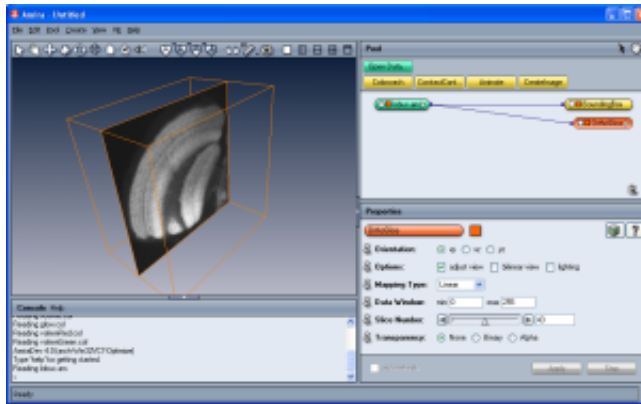
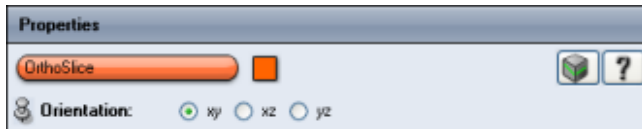


Figure 2.6: The *OrthoSlice* module is able to extract arbitrary orthogonal slices from a regular 3D scalar field or image volume.



If you want to remove a module permanently, select it and choose *Remove Object* from the *Pool* menu. Choose *Remove All* from the same menu to remove all modules.

- Remove the *BoundingBox* module by selecting its icon and choosing *Remove Object* from the *Pool* menu.
- Remove all remaining modules by choosing *Remove All Objects* from the same menu.

Now the Pool should be empty again. You may continue with the next tutorial, i.e., the one on *scalar field visualization*.

2.2 How to load image data

Loading image data is one of the most basic operations in *amira*. Other than with 2D images, there are not many standardized file formats containing 3D images. This tutorial guides you by means of examples on how to load the different kinds of 3D images into *amira*. In particular this tutorial covers the following topics:

1. Using the *File/Open Data...* browser and setting the file format.
2. Reading 3D image data from multiple 2D slices.

3. Setting the bounding box or voxel size of 3D images.
4. The *Stacked Slices* file format.
5. Working with LargeDiskData.

2.2.1 The amira file browser

Image data is loaded in amira with the *File/Open Data...* dialog. All *file formats* supported by amira are recognized automatically either by a data header or by the file name suffix. What follows is only of concern in these cases:

- The automatic file format detection fails.
- 3D image data is stored in several 2D files.
- The data is larger than the available main memory.

Setting the file format

In most cases the format of a file is determined automatically, either by checking the file header or by comparing the file name suffix with a list of known suffixes. In the load dialog the file format is displayed in a separate column in detail view.

Example:

- Files containing the string AmiraMesh in the first line are considered *AmiraMesh* files.
- Files with the suffix *.stl* are considered STL files.

If automatic file format detection fails, e.g. because some non-standard suffix has been used, the format may be set manually using the *Format* entry in the pop-up menu of the *Load* dialog (right mouse button).

2.2.2 Reading 3D image data from multiple 2D slices

A common way to store 3D image data is to write a separate 2D image file for each slice. The 2D images may be written in TIFF, BMP, JPEG, or any other supported image *file format*. In order to load such data in amira, all 2D slices must be selected simultaneously in the file browser. This can be done by clicking the first file and shift clicking the last one.

- Open the *File/Open Data...* dialog.
- Browse to the */Amira-4.1/data/multichannel1/channel/* directory.
- Select the first file *pvcca1.0001.jpeg*
- Shift-click the last file (*pvcca1.0048.jpeg*).
- Click *Load*.

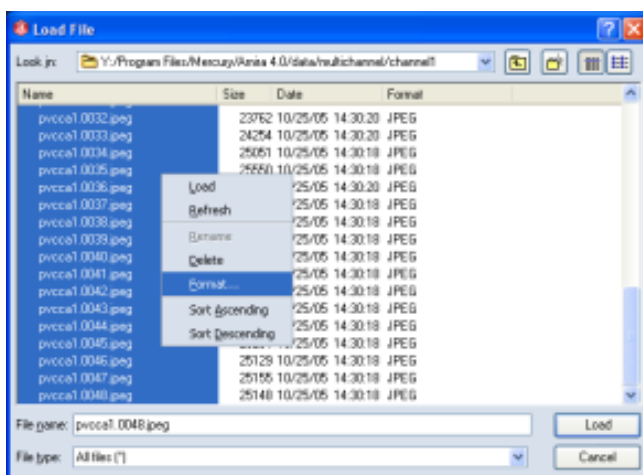


Figure 2.7: The *Format* option of the file browser

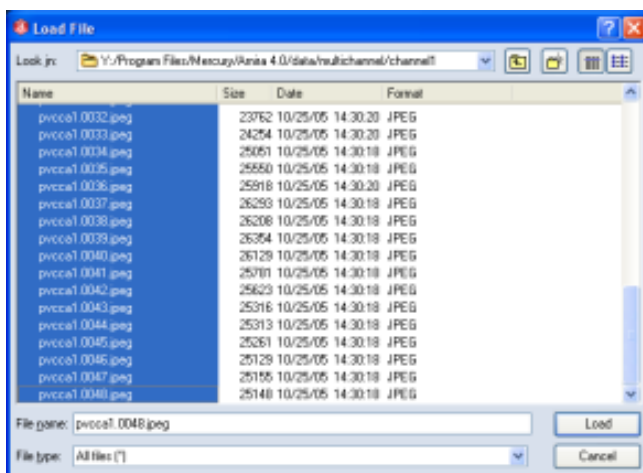


Figure 2.8: Loading multiple 2D images

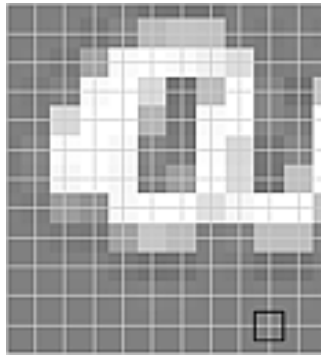


Figure 2.9: The definition of the bounding box in *amira*. Different gray shades depict the intensity values defined on the regular grid (white lines). The black square depicts the extent of one voxel. The outer frame depicts the extent of the bounding box.

2.2.3 Setting the bounding box

When loading a series of bitmap images, usually the physical dimensions of the images are not known to *amira*. Therefore an *Image Read Parameters* dialog appears that prompts you for entering the physical extent of the *bounding box*. Alternatively, the size of a single voxel can be set. In *amira* the bounding box of an object is the smallest rectangular, axis-aligned volume in 3D space that encompasses the object. *Note that in amira the bounding box of a uniform data set extends from the center of the first voxel to the center of the last one. For example, if you have 256 voxels and you know the voxel size to be 1 mm, the bounding box should be set to 0 - 255 (or to some shifted range).*

- Enter 0.85 in the first and second text fields and 3.5 in third text field of the *Voxel Size* port.
- Click *OK*.

This method will always create a data set with uniform coordinates, i.e., uniform slice distance. In case of variable slice distances, the *StackedSlices* format should be used.

2.2.4 The Stacked Slices file format

Especially with histological serial sections it often happens that slices are lost during preparation. To handle such cases, *amira* provides a special data type corresponding to a file format, called *Stacked Slices*. This file format allows a stack of individual image files to be read with optional *z*-values for each slice. The slice distance is not required to be constant. The images must be one-channel or RGBA images in an image format supported by *amira* (e.g. TIFF). The reader operates on an ASCII description file, which can be written with any editor. Here is an example of a description file:

```
# Amira Stacked Slices
```

```
# Directory where image files reside
pathname C:/data/pictures
# Pixel size in x- and y-direction
pixelsize 0.1 0.1
# Image list with z-positions
picture1.tif 10.0
picture7.tif 30.0
picture13.tif 60.0
colstars.jpg 330.0
end
```

Some remarks on the syntax:

- # Amira Stacked Slices is an optional header that allows amira to automatically determine the file format.
- pathname is optional and can be included if the pictures are not in the same directory as the description file. A space separates the tag "pathname" from the actual pathname.
- pixelsize is optional, too. The statement specifies the pixel size in x- and y-directions. The bounding box of the resulting 3D image is set from 0 to pixelsize*(number_of_pixels-1).
- picture1.tif 10.0 is the name of the slice and its z-value, separated by a space character.
- end indicates the end of the description file.
- Comments are indicated by a hash-mark character (#).

2.2.5 Working with Large Disk Data

Sometimes image data are so large that they do not fit into the main memory of the computer. Since the **amira** visualization modules rely on the fact that data are in physical memory, this would mean that such data cannot be displayed in **amira**. To overcome this, a special purpose module is provided that leaves most of the data on disk and retrieves only a user-specified subvolume. This subvolume can then be visualized with the standard visualization modules in **amira**.

- Use the *File/Open Data...* dialog and go to c:/Program Files/Amira-4.1/data/medical/
- Right-click on the reg005.ctdata.am and select the *Format* entry from the pop-up dialog
- Select *AmiraMesh as LargeDiskData* as format and confirm your choice with *OK*.
- Press the *Load* button.

The data will be displayed in the Pool as a regular green data icon. The info line indicates that it belongs to the data class *HxRawAsExternalData*.

- Right mouse click, attach a *BoundingBox* module.

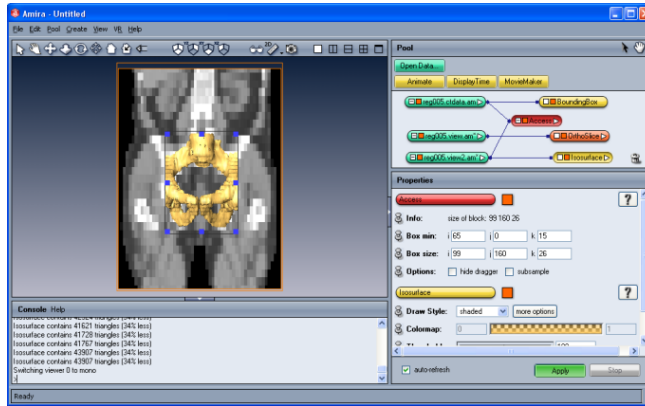


Figure 2.10: The usage of *AmiraMesh* as *LargeDiskData*. For instantaneous update, the *auto-refresh* check boxes of the *Access* and *Isosurface* modules have been checked

- Right mouse click, attach an *Access* module.
- Select the *Access* module in the Pool and enter 224, 161, and 59 into the *BoxSize* text fields.
- Check *Subsample* and enter 4 4 2 into the *Subsample* fields and press the *Apply* button.

This retrieves a down-sampled version of the data. Disconnect the *reg005.view.am* data icon from the *Access* module and use it as an overview (e.g. with *OrthoSlice*).

- Select the *Access* module in the Pool and deselect the *subsample* check box.
- Use the dragger box in the viewer to resize the subvolume.
- Press the *Apply* button.
- Attach an *Isosurface* module to the *reg005.view2.am* (set threshold set to 100).

Tip: To browse the data, check the *auto-refresh* check box for the *Access* and *Isosurface* modules. Now each time the blue subvolume dragger is repositioned, the visualization is updated automatically.

Loading *AmiraMesh*, *StackedSlices*, and *Raw "asLargeDiskData"* is a convenient and fast way of exploring data that exceed the size of system memory. However, especially with *StackedSlices*, it is not always the most efficient way of doing this. *amira* can store the image data in a special format that facilitates the random retrieval of data from disk.

- Choose from the *Create/Data* menu *ConvertToLargeDiskData*
- Click *Browse* from the *Input* port.
- Click *Add*, go to */Amira-4.1/data/medical/* and select *reg005.ctdata.am*, then click *Load*.
- Click *Browse* from the *Output* port.

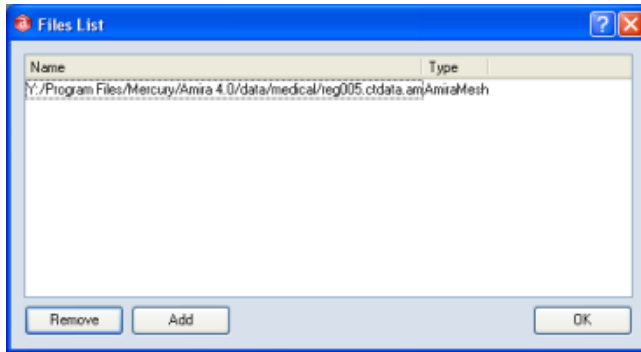


Figure 2.11: The *Input* dialog of the *ConvertToLargeDiskData* module.

- Go to `C : /tmp/` and enter a filename of your choice.
- Press the *Apply* button.

Although you will most likely not notice any difference with the small image data used in this tutorial, this method for retrieving large data significantly accelerates the *Apply* operation.

2.3 Visualizing 3D Images

This section provides a step-by-step introduction to the visualization of regular scalar fields, e.g., 3D image data. *amira* is able to visualize more complex data sets, such as scalar fields defined on curvilinear or tetrahedral grids. Nevertheless, in this section we consider the simplest case, namely scalar fields with regular structure. Each step builds on the step before. In particular, the following topics will be discussed:

1. orthogonal slices
2. simple threshold segmentation
3. resampling the data
4. displaying an isosurface
5. cropping the data
6. volume rendering

We start by loading the data you already know from Section 2.1 (Getting Started): a 3D image data set of a part of a fruit fly's brain. The data set has been recorded with a confocal laser scanning microscope at the University of Wuerzburg.

- Load the file `lobus.am` located in subdirectory `data/tutorials`.

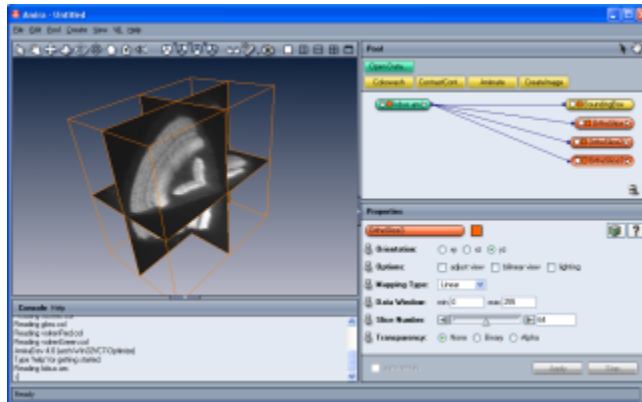


Figure 2.12: Lobus data set visualized using three orthogonal slices.

2.3.1 Orthogonal Slices

The fastest and in many cases most “standard” way of visualizing 3D image data is by extracting orthogonal slices from the 3D data set. *amira* allows you to display multiple slices with different orientations simultaneously within a single viewer.

- Connect a *BoundingBox* module to the data (use right mouse on lobus.am).
- Connect an *OrthoSlice* module to the data.
- Connect a second and third *OrthoSlice* module to the data.
- Select *OrthoSlice2* and press *xz* or *coronal* in the *Orientation* port.
- Similarly, for *OrthoSlice3* choose *yz* or *sagittal* orientation.
- Rotate the object in the viewer to a more general position.
- Change the slice numbers of the three *OrthoSlice* modules in the respective ports or directly in the viewer as described in section Getting Started.

In addition to the *OrthoSlice* module, which allows you to extract slices orthogonal to the coordinate axes, *amira* also provides a module for slicing in arbitrary orientations. This more general module is called *ObliqueSlice*. You might want to try it by selecting it from the Display submenu of the lobus data popup menu.

2.3.2 Simple Data Analysis

The values of the *data window* port of the *OrthoSlice* module determine which scalar values are mapped to black or white, respectively. If you choose a range of e.g., 30...100, any value smaller or equal to 30 will become black, and all pixels with an associated value of more then 100 will become

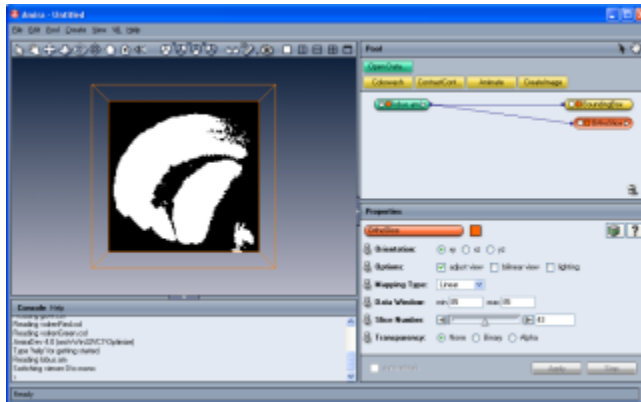


Figure 2.13: By adjusting the data window of the *OrthoSlice* module a suitable value for threshold segmentation can be found. Intensity values smaller than the min value will be mapped to black, intensity values bigger than the max value will be mapped to white.

white. Try modifying the range. This port provides a simple way of determining a threshold, which later can be used for segmentation, e.g., in biology or medicine to separate background pixels from anatomical structures. This can be most easily done by making the minimum and maximum values coincide.

- Remove two of the *OrthoSlice* modules.
- Select the remaining *OrthoSlice* module.
- Make sure that the *mapping type* is set to *linear*.
- Change the minimum and maximum values of the data window port until these values are the same and a suitable segmentation result is obtained. For this data set 85 should be a good threshold value.

A more powerful way of quantitatively examining intensity values of a data set is to use a data probing module *PointProbe* or *LineProbe*. However, we will not discuss these modules in this introductory tutorial.

2.3.3 Resampling the Data

Now we are going to compute and display an *isosurface*. Before doing so, we will resample the data. The resampling process will produce a data set with a coarser resolution. Although this is not necessary for the isosurface tool to work, it decreases computation time and improves rendering performance. In addition, you will get acquainted with another type of module. The *Resample* module is a computational module. Computational modules are represented by red icons. Typically you must press the green *Apply* button at the bottom of the Properties Area to start the computation. After you

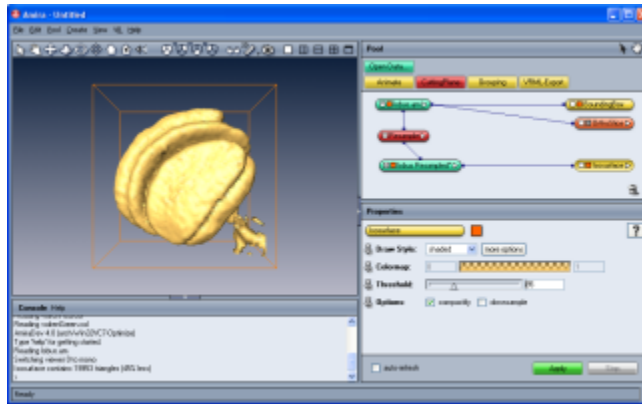


Figure 2.14: Lobus data set visualized in 3D using an isosurface.

press this button they produce a new data object containing the result.

- Connect a *Resample* module to the data and select it.
- Enter values for a coarser resolution, e.g., $x=64$, $y=64$, $z=43$.
- Press the *Apply* button.

A new green data icon representing the output of the resample computation named *lobus.Resampled* is created. You can treat this new data set like the original lobus data. In the popup menu of the resampled lobus you will find exactly the same attachable modules and you can save and load it like the original data.

You may want to compare the resampled data set with the original one using the *OrthoSlice* module. You can simply pick the blue line indicating the data connection and drag it to a different data source. Whenever the mouse pointer is over a valid source, the connection line appears highlighted in lighter blue.

2.3.4 Displaying an Isosurface

For 3D image data sets, isosurfaces are useful for providing an impression of the 3D shape of an object. An isosurface encloses all parts of a volume that are brighter than some user-defined threshold.

- Turn off the viewer toggle of the *OrthoSlice* module.
- Connect an *Isosurface* module to the resampled data record and select it.
- Adjust the threshold port to 85 or a similar value.
- Press the *Apply* button.

2.3.5 Cropping the Data

Cropping the data is useful if you are interested in only a part of the field. A crop editor is provided for this purpose. Its use is described below:

- Remove the resampled data *lobus.Resampled*.
- Activate the display of the *OrthoSlice* module.
- Select the *lobus.am* data icon.
- Click on the Crop Editor button in the Properties Area.

A new window pops up. There are two ways to crop the data set. You can either type the desired ranges of x, y, and z coordinates into the crop editor's window or put the viewer into interaction mode and adjust the crop box using the green handles directly in the viewer window.

- Put the viewer into interaction mode.
- With the left mouse button, pick one of the green handles attached to the crop volume. Drag and transform the volume until the part of the data you are interested in is included.
- Press *OK* in the crop editor's dialog window.

The new dimensions of the data set are given in the Properties Area. If you want to work with this cropped data record in later sessions you should save it by choosing *Save Data As ...* from the *File* menu.

As you already might have noticed, the crop editor also allows you to rescale the bounding box of the data set. By changing the bounding box alone, no voxels will be cropped. You may also use the crop editor to enlarge the data set, e.g., by entering a negative value for the *k min* number. In this case the first slice of the data set will be duplicated as many times as necessary. Also, the bounding box will be updated automatically.

2.3.6 Volume Rendering

Volume Rendering is a visualization technique that gives a 3D impression of the whole data set without segmentation. The underlying model is based on the emission and absorption of light that pertains to every voxel of the view volume. The algorithm simulates the casting of light rays through the volume from pre-set sources. It determines how much light reaches each voxel on the ray and is emitted or absorbed by the voxel. Then it computes what can be seen from the current viewing point as implied by the current placement of the volume relative to the viewing plane, simulating the casting of sight rays through the volume from the viewing point.

You can choose between two different methods for these computations: *maximum intensity* projections or an ordinary emission-absorption model.

- Remove all objects in the Pool other than the *lobus.am* data record.

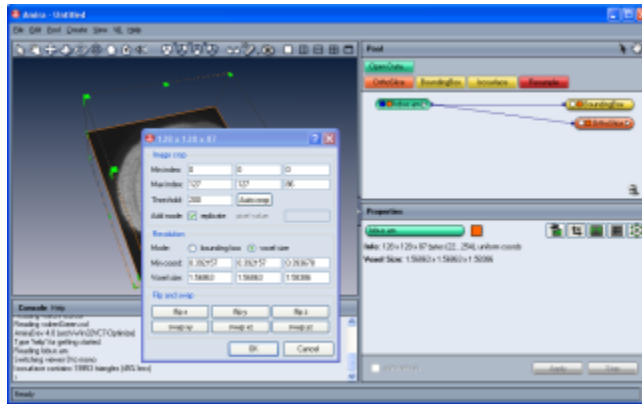


Figure 2.15: The crop editor works on uniform scalar fields. It allows you to crop a data set, to enlarge it by replicating boundary voxels, or to modify its coordinates, i.e. to scale or shift its bounding box.

- Connect a *Vortex* module to the data.
- Select the data icon and read off the range of data values printed on the first info line (22...254).
- Select the *Vortex* module and enter the range in the *Colormap* port.
- Press the *Apply* button in order to perform some texture preprocessing which is necessary for visualizing the data.

By default, emission-absorption volume rendering is shown. The amount of light being emitted and absorbed by a voxel is taken from the color and alpha values of the colormap connected to the *Vortex* module. In our example the colormap is less opaque for smaller values. You may try to set the lower bound of the colormap to 40 or 60 in order to get a better feeling for the influence of the *transfer function*. In order to compute *maximum intensity* projections, choose the *mip* option of port *Options*.

Internally, the *vortex* module makes heavy use of OpenGL texture mapping. Both texture modes, 2D and 3D, are implemented. 3D textures yield slightly better results. However, this mode is not supported by all graphics boards. The 3D texture mode requires you to adjust the number of slices cut through the image volume. The higher this number the better the results are.

Alternatively, 2D textures can be used for volume rendering. In this case, slices perpendicular to the major axes are used. You may observe how the slice orientation changes if you slowly rotate the data set. The 2D texture mode is well suited for mid-range graphics workstations with hardware accelerated texture mapping. If your computer does not support hardware texture mapping at all, you should use visualization techniques other than volume rendering.

- Make sure the *mip* button of port *Options* is unchecked.
- If 3D texture mode is enabled, choose about 200 slices.
- Click with the right mouse button on port *Colormap* and choose *volrenRed.icol*.

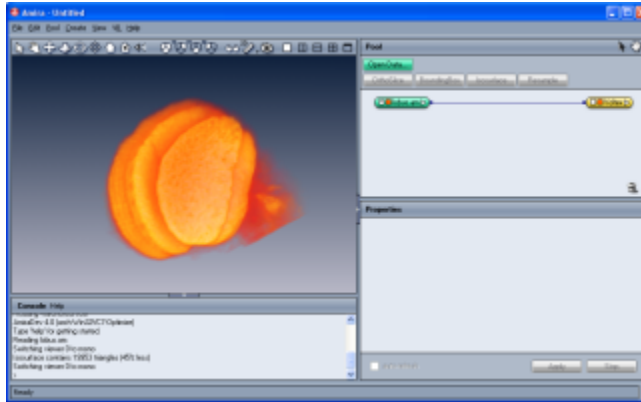


Figure 2.16: The *Voltex* module can be used to generate maximum intensity projections as well as volume renderings based on an emission-absorption model. In both cases, 2D or 3D texture mapping techniques can be applied.

- Set *Lookup* to RGBA and change the min and max values of the colormap to 40 and 150.
- Finally, press *Apply* in order to initialize the *Voltex* textures.

Whenever you choose a different colormap or change the min and max values of the colormap, you must press the *Apply* button again. This causes the internal texture maps to be recomputed. An exception are SGI systems with Infinite Reality graphics. On these platforms a hardware-specific OpenGL extension is exploited, causing colormap changes to take effect immediately.

2.4 Segmentation of 3D Images

By following this step-by-step tutorial you will learn how to interactively create a segmentation of a 3D image. A segmentation assigns to each pixel of the image a label describing to which region or material the pixel belongs, e.g., bone or the kidney. The segmentation is stored in a separate data object called a *LabelField*. A segmentation is the prerequisite for surface model generation or accurate volume measurement.

This tutorial comprises the following steps:

1. Creation of an empty *LabelField*.
2. Interactive editing of the labels in the *Image Segmentation Editor*.
3. Measuring the volume of the segmented structures.
4. An alternative segmentation method: Threshold segmentation.

2.4.1 Interactive Image Segmentation

- Load the *lobus.am* data file from the directory *data/tutorials*.
- Right click on the green icon and choose *LabelField* from the Labelling section.

A new green icon appears, the *LabelField* that will hold the segmentation results. Simultaneously, the image segmentation editor is displayed in the 3D viewer pane.

By default, the segmentation editor operates in 4-viewer mode. In this mode, three 2D viewers with different orientations and an additional 3D viewer are displayed.

The tools of the segmentation editor are displayed where the Pool and Properties Area are normally displayed. You can click on the tabs at the top of this region to switch between the two displays. For this tutorial, you'll want to display the segmentation tools.

- Use the mouse to expand the viewer so that you have more room to maneuver in.
- In the lower right view, use the slider on the bottom to scroll through the slices. Go to slice 20. You see two bigger structures and one structure just appearing on the top.
- If necessary, click on the second button under the label *Zoom and Data Window* to zoom out the data so that you have a view of the entire slice.
- Click on the second button under the label *Tools*, the brush.
- Mark the rightmost structure with the mouse. Hold down the control button to unselect wrongly selected pixels if necessary.
- When done, select the entry *Inside* in the *Materials* list. Then click the + button under the *Selection* label.

The previously selected pixels are now assigned to the material *Inside*. You can right click on the entry *Inside* in the *Materials* list and choose a different draw style (for example, dotted).

- Click into the material list and choose *New Material* from the right button menu.
- Mark the middle structure using the brush, select the new material in the *Materials* list and assign the pixels to that structure.
- Go to slice 21 and practice by segmenting the two structures.

Hint: If you prefer to work with one larger view rather than four smaller views, click on the *Layout1* button in the viewer toolbar. To cycle through each of the four views, press the *Layout1* button repeatedly. To return to 4-viewer mode, press the *Layout4* button.

If a structure does not change a lot from slice to slice, you can use interpolation.

- Go to slice 22 and mark the right structure using the brush. Go to slice 31 and mark the same structure.
- Choose from the menu bar: *Selection/Interpolate*.
- Scroll through the data set. You should see that the in between slices 23 to 30 are selected too.

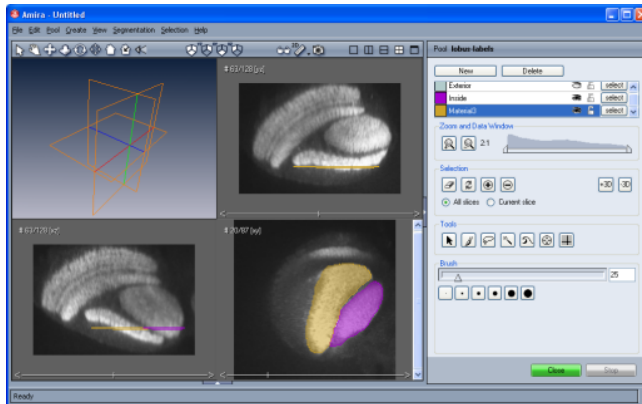


Figure 2.17: Image segmentation editor after selecting and assigning pixels for two structures in one slice

- In order to assign the selected pixels in all slices to the Inside material, select the Inside material in the list, then click the + button.
- Repeat the procedure between slice 32 and 50.
- Repeat the procedure for the middle structure.

Hints:

- It is highly recommended to frequently save the segmentation results while working. In order to do so, select the label field in the **amira** main window and choose *Save* or *Save As...* from the **amira** File menu.
- The brush is only the most basic segmentation tool. The segmentation editor provides many more functions that are described on its *reference page*.
- There are many useful key bindings, including **SPACE** and **BACKSPACE** to change the slice number or "d" to toggle the draw style.
- Of course you can give the materials more meaningful names or colors using the context menu (right mouse button in the list).

At this point you may want to close the editor by pressing the *Close* button. Save the label field. In the next tutorial you will learn how to create a 3D surface model from the segmentation results.

2.4.2 Volume Measurement

Once a structure is segmented, you can easily measure its volume:

- Right click on the LabelField's green icon. Choose *Measure/TissueStatistics*.

- Press the *Apply* button. A new icon appears.
- Select this icon and press the *Show* button.

The units in the volume column depend on the units you have specified the voxel size. In case of the *lobus.am*, the voxel size is in μm , therefore the volume is in μm^3 .

2.4.3 Threshold Segmentation

We now describe an alternative way of segmentation, which can require less manual interaction, but only works for images with good quality.

In some cases a satisfying segmentation can be achieved automatically based solely on the gray values of the image data set.

The first step is to separate the object from the background. This is done by segmenting the volume into exterior and interior regions on the basis of the voxel values.

- Load the *lobus.am* data record from the directory *data/tutorials*.
- Attach a *LabelVoxel* module to the data icon and select it.
- Type 85 into the text field of port *Exterior-Inside*. You may also determine some other threshold that separates exterior and interior as described in the tutorial on Image Data Visualization.
- Press the *Apply* button.

By this procedure each voxel having a value lower than the threshold is assigned to *Exterior* and each voxel whose value is greater than or equal to the threshold is assigned to *Interior*. This may, however, cause artifacts that are not part of the object but which have voxel values above the threshold to be assigned to the interior. This can be suppressed by setting the *remove couch* option which assures that only the biggest coherent area will be labeled as the interior and all other voxels are assigned to the exterior.

After you press the *Apply* button, a new data object is computed and its icon appears in the Pool. The data object is denoted *lobus.Labels*. It is of type *LabelField*, represents a cubic grid with the same dimensions as *lobus.am*, and contains an interior or exterior label for each voxel according to the segmentation result.

2.4.4 Refining Threshold Segmentation Results

You can visualize and manually modify a *LabelField* by using *amira's image segmentation editor*. A more detailed description of this tool is contained in the User's Reference guide. Here, we use the image segmentation editor to smooth the data in order to get a nicer looking surface of the object.

- Select the *lobus.Labels* icon and click on the Image Segmentation icon in the green title bar in the Properties Area.

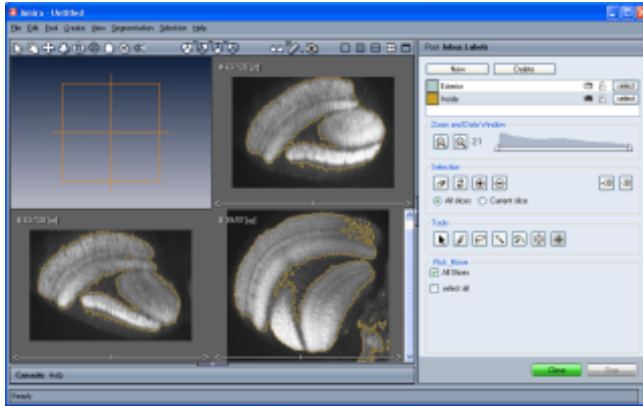


Figure 2.18: Data from confocal microscopy is segmented using amira's image segmentation editor.

In response the image segmentation editor is popped up.

- In the lower right view, use the slider on the bottom to select slice 39.
- Choose a magnification ratio of 2:1 by pressing the zoom-up button in the Zoom and Data region of the editor.

The image segmentation editor shows the image data to be segmented (*lobus.am*) as well as brownish contours representing the borders between interior and exterior regions as contained in the *lobus.Labels* data object. As you can see, the borders are not so smooth and there are many little islands, bordered by brownish contours. This is what we want to improve now.

- Choose *Remove Islands* from the editor's *Segmentation* menu. In response, a little dialog window appears.
- In the dialog window select the *all slices* mode. Then press *Remove* in order to apply the filter in all slices. Note how the segmentation results become less noisy.
- To further clean up the image, choose *Smooth Labels* from the editor's *Segmentation* menu. Another dialog box appears.
- Select the *3D volume* mode and press the *Apply* button in order to execute the smoothing operation.
- To examine the results of the filter operations, browse through the label field slice by slice. In addition to the slice slider you may also use the cursor-up and cursor-down keys for this.
- Click on the *Close* button to close the image segmentation editor.

2.5 Surface Reconstruction from 3D Images

By following this step-by-step tutorial, you will learn how to generate a triangular surface grid for an object embedded in a voxel data set. A surface grid allows for producing a 3D view of the object's surface and can be used for numerical simulations.

The generation process consists of these steps:

1. Extracting surfaces from segmentation results
2. Simplifying the surface

As a prerequisite for the following steps, you need a label field, which holds the result of a previous image segmentation. You can either use the label field which you created in the previous tutorial or load the provided *lobus.labels* data set from the *data/tutorials* directory.

2.5.1 Extracting Surfaces from Segmentation Results

Now we let *amira* construct a triangular surface of the segmented object.

- Connect a *SurfaceGen* module to the *lobus.Labels* data.
- Press the *Apply* button.

The option *add border* ensures that the created surface be closed. A new data object *lobus.surf* is generated. Again, it is represented by a green icon in the Pool.

2.5.2 Simplifying the Surface

Usually the number of triangles created by the *SurfaceGen* module is far too large for subsequent operations. Thus, the number of triangles must be reduced in a *surface simplification* step. In *amira* a *Surface Simplification Editor* is provided for this purpose.

- Select the surface *lobus.surf*.
- Click on the Simplifier button (triangle mesh icon) in the Properties Area.
- Set the desired number of faces to 3500 in the *Simplify* port.
- Turn on the *fast* toggle in the *Options* port. This option disables some time-consuming intersection tests.
- Push the *Simplify now* button in the *Action* port.

The number of triangles is reduced to about 3500 now. The progress bar tells you how much of the simplification task has already been done.

To examine the simplified surface, attach a *SurfaceView* module to the *lobus.surf* data object.

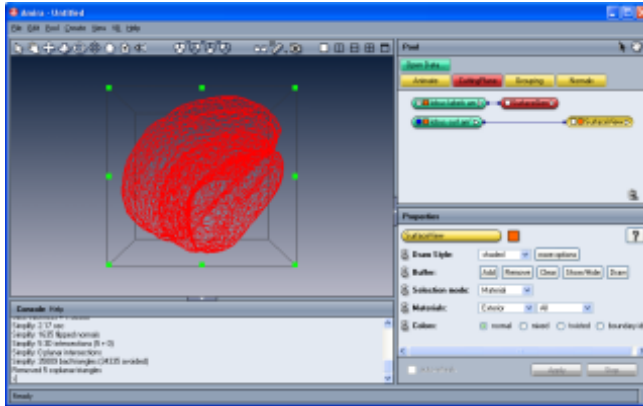


Figure 2.19: Surface representation of optical lobus as triangular grid

The *SurfaceView* module maintains an internal buffer and displays all triangles stored in this buffer. By default the buffer shows all triangles forming the boundary to the exterior. If you change the selection at the *Materials* port, the newly selected triangles are highlighted, i.e., they are displayed using a red wireframe representation. The *Add* and *Remove* buttons cause the highlighted triangles to be added to or removed from the buffer, respectively. You may easily visualize a subset of all triangles using a 3D selection box or by drawing contours in the 3D viewer. Press the *Clear* button of the *Buffer* port to see the display shown in Figure 2.19.

2.6 Creating a Tetrahedral Grid from a Triangular Surface (Mesh Pack)

By following this step-by-step tutorial, you will learn how to generate a volumetric tetrahedral grid from a triangular surface as created in the previous tutorial. A tetrahedral grid is the basis for producing various views of inner parts of the object, e.g., cuts through it, and is frequently used for numerical simulations.

The generation process consists of these steps:

1. Simplifying the surface
2. Editing the surface
3. Generating a tetrahedral grid

As a prerequisite for the following steps, you need a triangular surface, which is usually the result of a previous surface reconstruction. Load the supplied *lobus.surf* data set from the *data/tutorials* directory.

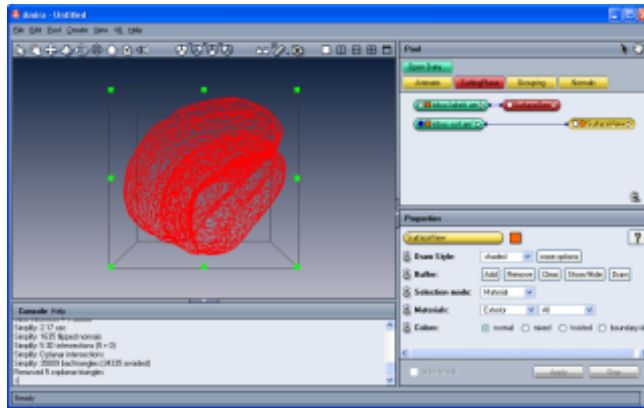


Figure 2.20: Surface representation of optical lobe as triangular grid

2.6.1 Simplifying the Surface

Usually the number of triangles created by the *SurfaceGen* module is far too large for subsequent operations, e.g., for a numerical simulation. Thus, the number of triangles should be reduced in a *surface simplification* step. In *amira* a *Surface Simplification Editor* is provided for this purpose. There may be different goals for the simplification:

- In *computer graphics*, one wants to prescribe just the number of faces, because this determines the rendering speed.
- For a *numerical simulation*, one often wants to specify the maximum edge length occurring in the grid model.

This tutorial shows how the maximum edge length can be controlled during simplification.

- Select the surface *lobus.surf*.
- Click on the Simplifier (triangle mesh icon) in the Properties Area.
- Set the desired number of faces to 1000 and the desired maximal distance (i.e. edge length) to 10 in the *Simplify* port.
- Leave the *fast* toggle turned off in the *Options* port. This will cause intersection tests to be performed during simplification, which will considerably reduce the probability that the simplified surface contains self intersections.
- Press the *Simplify now* button in the *Action* port.

Simplification terminates when either of the limits given by the number of faces or the maximum distance is reached. The progress bar tells you how much of the simplification task has already been done. In this example the maximum distance will be the limiting factor, and the resulting surface will

contain about 6000 faces.

Besides the maximum edge length, the minimum edge length occurring in the surface should also be controlled, because the ratio of maximum and minimum edge length will influence the quality of the resulting tetrahedral grid. This ratio should not be much larger than 10. If edges that are too short occur in the simplified surface, they can be removed as follows.

- Set the desired minimum distance to 2 in the *Simplify* port.
- Observe the number of faces as shown at the *Surface* port, and press the *Contract edges* button in port *Action*. All edges shorter than 2 will be contracted. In this example about 30 small edges will be detected. You will observe that the number of faces slightly decreases.

2.6.2 Editing the Surface

As a second step of preparation for tetrahedral grid generation, invoke the *Surface Editor*.

- Select the surface *lobus.surf*.
- Leave the *Surface Simplification Editor* (Simplifier) by again clicking on the triangle mesh icon.
- Enter the *Surface Editor* by clicking on the *Surface Editor* button in the Properties Area.

Automatically, a *SurfaceView* module will be attached to the *lobus.surf* surface. For details about that module see its description.

When the *Surface Editor* is invoked, the *Surface* menu is added to *amira*'s menu bar and a new toolbar is placed just below *amira*'s viewer toolbar. The *Surface/Tests* menu contains 6 specific tests which are useful for preparing a tetrahedral grid generation. Each of the tests creates a buffer of triangles which can be cycled through using the back and forward buttons.

- Select *Intersection test* from the *Surface/Tests* menu. The total number of intersecting triangles is printed in the console window. Intersections shouldn't occur too often if toggle *fast* was switched off during surface simplification. In case they occur, the first of the intersecting triangles and its neighbors are shown in the viewer window.
- You can manually repair intersections using four basic operations: *Edge Flip*, *Edge Collapse*, *Edge Bisection*, and *Vertex Translation*. See the description of the *Surface Editor* for details.
- After repairing, invoke the intersection test again by selecting it from the *Surface/Tests* menu or by pressing the *Compute* button.
- When the intersection test has been successfully passed, select the *Orientation test* from the *Surface/Tests* menu. After surface simplification, the orientation of a small number of triangles may be inconsistent, resulting in a partial overlap of the materials bounded by the triangles. In case of such incorrect orientations, which should occur quite rarely, there is an automatic repair. If this fails, the detected triangles will be shown, and you can use the above mentioned manual operations for repair. **Note:** There are two prerequisites for the orientation test: the surface must be free of intersections, and the outer triangles of the surface must be assigned to material

Exterior. If the surface does not contain such a material or if the assignment to *Exterior* is not correct, the test will falsely report orientation errors.

A successful pass of the intersection and orientation test is mandatory for tetrahedral grid generation. These tests are automatically performed at the beginning of grid generation. So you can directly enter the *TetraGen* module (see below) and try to create a grid. If one of the tests fails, an error message will be issued in the console window. You can then go back to the *Surface Editor* and start editing.

The remaining three tests analyze the surface mesh with respect to different quality measures. These tests only need to be performed if the tetrahedral quality of the volumetric grid plays an important role, e.g., if the grid will be used for a numerical simulation.

- Select *Aspect ratio* from the *Surface/Tests* menu. This computes the ratio of the radii of the circumcircle and the incircle for each triangle. The triangle with the worst (i.e. largest) value is shown first, and the actual value is printed in the console window. The largest aspect ratio should be below 20 (better below 10). Fortunately there is an automatic tool for improving the aspect ratio included in the *Surface Editor*.
- Select *Flip edges* from the *Edit* menu. A small dialog window appears. In the Radius Ratio area, set the value of the "Try to flip a triangle..." field to 10. Select mode *operate on whole surface*. Press button *Flip*. All triangles with an aspect ratio larger than 10 will be inspected; if the aspect ratio can be improved via an edge flip, this will be done automatically. The console window will tell you the total number of bad triangles and how many of them could be repaired. Press the *Close* button to leave the *Flip edges* tool.
- Select again *Aspect ratio* from the *Surface/Tests* menu. Only a small number of triangles with large aspect ratio should remain after applying the *Flip edges* tool.
- Select *Dihedral angle* from the *Surface/Tests* menu. For each pair of adjacent triangles, the angle between them at their common edge will be computed. The triangle pair including the worst (i.e. smallest) angle is shown in the viewer, and the actual value is printed in the console window. The smallest dihedral angle should be larger than 5 degrees (better larger than 10).
- For a manual repair of a small dihedral angle proceed as follows: select the third points of both triangles (i.e. the points opposite to the common edge) and move them away from each other. For moving vertices you must enter *Vertex Translation* mode by clicking on the first icon from the right on the top of the viewer window or by pressing the "t" key. If the viewer is in viewing mode, switch it into interaction mode by pressing the ESC key or by clicking on the arrow icon (the first icon from the top) on the right of the viewer window. Click on the vertex to be moved. At the picked vertex a point dragger will be shown. Pick and translate the dragger for moving the vertex.
- In some cases an edge flip might also improve the situation. Enter *Edge Flip* mode by clicking on the third icon from the right on the top of the viewer window or by pressing the "f" key. Switch the viewer into interaction mode. Click on the edge to be flipped.
- Select *Tetra quality* from the *Surface/Tests* menu. For each surface triangle the aspect ratio of the tetrahedron which would probably be created for that triangle will be calculated. The aspect

ratio for a tetrahedron is defined as the ratio of the radii of the circumsphere and the inscribed sphere. The triangle with the worst (i.e. largest) value is shown in the viewer, and the actual value is printed in the console window. The largest tetrahedral aspect ratio should be below 50 (better below 25). If all small dihedral angles have already been repaired, the tetra quality test will mainly detect configurations where the normal distance between two triangles is small compared to their edge lengths. Again, the *vertex translation* and the *edge flip* operation are best suited for a manual repair of large tetrahedron aspect ratios.

- Leave the *Surface Editor* by again clicking on the Surface Editor button in the Properties Area.

Hint: In order to see the entire surface again, select the SurfaceView icon, then press its *Show/Hide* button, then press the *ViewAll* button in the viewer toolbar.

2.6.3 Generation of a Tetrahedral Grid

The last step is the generation of a *volumetric tetrahedral grid* from the surface. This means that the volume enclosed by the surface is filled with tetrahedra.

Because the computation of the tetrahedral grid may be time consuming it can be performed as a *batch job*. You can then continue working with *amira* while the job is running. However, for demonstration purposes we want to compute the grid right inside *amira*.

- Connect a *TetraGen* module to the *lobus.surf* surface by choosing *Compute TetraGen* from the popup menu over the *lobus.surf* icon.
- Leave toggle *improve grid* switched on and toggle *save grid* switched off at the *Options* port. The *improve grid* option will invoke an automatic post-processing of the generated grid, which improves tetrahedral quality by some iterations that move inner vertices and flip inner edges and faces. See the description of the *Grid Editor* for details.

If toggle *save grid* is selected, an additional port *Grid* appears, where you can enter a filename. The resulting tetrahedral grid will be stored automatically under that name. If you want to run grid generation as a batch job, you must select the *save grid* option.

- Press the *Meshsize* button of the *Action* port. An editor window will appear. It allows you to define a desired mesh size, i.e., mean length of the inner edges to be created, for each region. For this you must enter the bundle of that region, and select parameter *MeshSize*. Then you can change the value in the text field at the lower border of the editor. There are some predefined region names in *amira* for which a default mesh size will be automatically set. Make sure that the default values are suitable for your application. If you are not sure about a suitable value, set the desired mesh size to 0. In this case the mean edge length of the surface triangles will be used.
- Press the *Run now* button at port *Action*. A pop-up dialog appears asking you whether you really want to start the grid generation. Click *Continue* in order to proceed.

Once grid generation is running, the progress bar informs you about the number of tetrahedra which

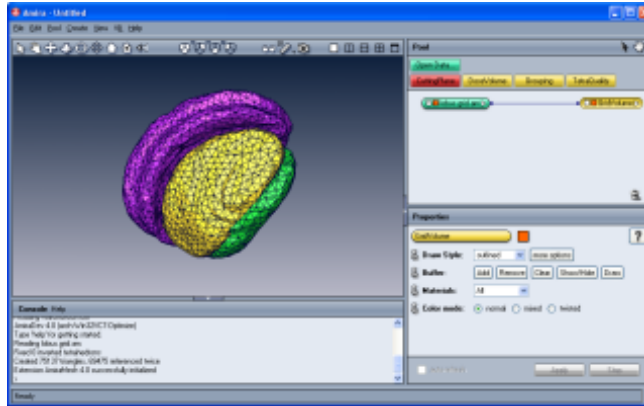


Figure 2.21: Volumetric representation of optical lobe as tetrahedral grid

already have been created. In some situations grid generation may fail, for example, if the input surface intersects itself. Then an error message will occur at the *Console Window*. In this case go back to the *Surface Editor* to interactively fix any intersections.

After the tetrahedral grid has been successfully created, a new icon called *lobus.grid* will be put in the Pool. You can select this icon in order to see how many tetrahedra the created grid contains. If grid generation takes too long, you may also load the pre-computed grid *lobus.grid* from the *data/tutorials* directory.

As the very last step you may want to have a look at the fruits of your work:

- Attach a *GridVolume* module to the *lobus.grid*.
- Select the *GridVolume* icon and press the *Add to* button of the *Buffer* port.

The *GridVolume* module maintains an internal buffer and displays all tetrahedra stored in this buffer. By default the buffer is empty, but all tetrahedra are highlighted, i.e., they are displayed using a red wireframe representation. The *Add to* button causes the highlighted tetrahedra to be added to the buffer. You may easily visualize a subset of all tetrahedra using a 3D selection box or by drawing contours in the 3D viewer.

Similar to the *Surface Editor*, there is a *Grid Editor* which can be invoked by selecting the green icon of the tetrahedral grid and clicking on the pencil icon (first from the right in the title bar) in the Properties Area. The editor allows for selecting tetrahedra with respect to different quality measures, e.g., aspect ratio, dihedral angles at tetrahedron edges, solid angles at tetrahedron vertices, and edge length. The editor contains several modifiers that can be applied for improving mesh quality.

2.7 Warping and Registration Using Landmarks

This is an advanced tutorial. You should be able to load files, interact with the *3D viewer*, and be familiar with the 2-viewer layout and the viewer toggles.

We will transform two 3D objects into each other by first setting landmarks on their surfaces and then defining a mapping between the landmark sets. As a result we shall see a rigid transformation and a warping which deforms one of the objects to match it with the other. The steps are:

1. Displaying data sets in two viewers.
2. Creating a landmark set.
3. Alignment via a rigid transformation.
4. Warping two image volumes.

2.7.1 Displaying Data Sets in Two Viewers

The data we will be working with in this tutorial are of the same kind you have already seen before: Two optical lobes of a drosophila's brain.

- Load the two lobes by executing the script `share/examples/landmark.hx`.

This script will load two data sets called *lobus.am* and *lobus2.am*. In addition, two isosurface modules connected to each of the data sets will be created. In the viewer the two lobes are visualized by isosurfaces, the first in yellow and the second in blue. As we can see, the lobes are oriented differently. We want to look at each lobe in its own viewer.

- Choose *2 Viewers horizontal* from the *View Layout* menu.

You can see the two lobes in both viewers.

- Visualize the first lobe (blue) in the upper viewer and the second lobe (yellow) in the lower viewer. This is done by deactivating the lower viewer toggle (orange buttons in the icons) of the *Isosurface* module and by deactivating the upper viewer toggle in the *Isosurface2* module.

2.7.2 Creating a Landmark Set

Now, let us create a landmark set object.

- From the main window's *Create* menu select *Data/Landmarks (2 sets)*

in order to create an empty landmark object. A new green icon will show up in the Pool. Since we are going to match two objects by means of corresponding landmarks we had to select the landmark objects containing 2 sets of landmarks (*Landmarks (2 sets)*).

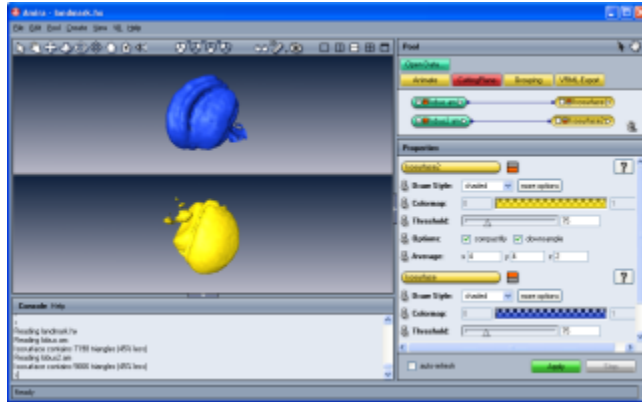


Figure 2.22: Two lobes visualized with isosurfaces in 2-viewer layout.

- Select object *Landmarks*.
- Launch the *Landmark Editor* by clicking on the Landmark Editor button in the Properties Area.

When starting the editor, a *LandmarkView* module is automatically created and connected to the *Landmarks* data object. As indicated on the info line, two empty landmark sets are available now. We use the editor to define some markers in both objects. For the following, it is useful to push-pin the three ports on the landmark editor. In order to do so, select the gray push-pin toggles to the left of the port labels.

- Connect a second *LandmarkView* module to the *Landmarks* object.
- Select the first *LandmarkView* module and choose *Point Set: Point Set 1*.
- Shift-select the second *LandmarkView* module and choose *Point Set: Point Set 2*.
- Set the viewer toggles of the two *LandmarkView* modules in the same manner as described for the *SurfaceView* modules previously. *LandmarkView* should be visible in the upper viewer and *LandmarkView2* in the lower viewer.

Before starting to set landmarks it is helpful to rotate the two lobes in their viewers such that they are approximately aligned. This will make it easier to locate corresponding features in the two objects and to select reasonable positions for landmarks.

- Rotate the two lobes to align them roughly.

Now, we are ready to define and set corresponding landmarks. Select the *Landmarks* object if necessary and choose the option

Edit mode: Add.

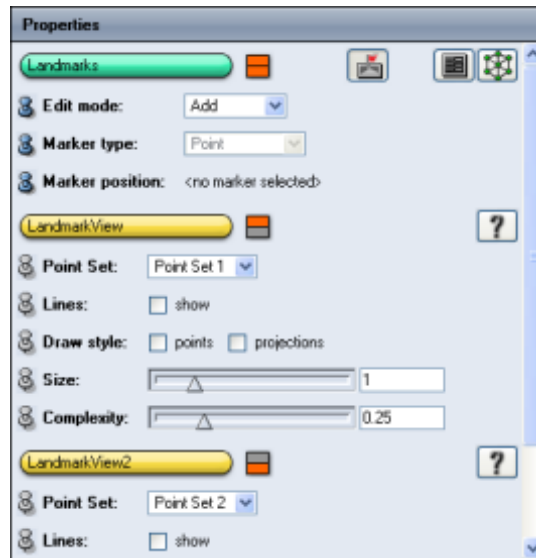


Figure 2.23: The image shows how the viewer toggles and *Point Set* ports should be set.

To set corresponding landmarks simply click with the left mouse button anywhere on the surface in the first viewer first and click on the surface in the second viewer subsequently. The landmarks are visualized as small spheres, the first landmark in yellow and the corresponding landmark in blue. Make sure to always set the first landmark on the first (yellow) surface and the second landmark on the second (blue) surface!!

If you want to change the position of an existing landmark set

Edit mode: Move

and select the respective landmark (blue or yellow) by clicking on it with the left mouse button. Then just click at the desired position.

You can also delete existing landmarks by setting

Edit mode: Remove

and clicking on the respective landmark. Both corresponding landmarks (blue and yellow) will be removed, no matter which one was selected.

You should now be able to create several landmarks. You may want to change the view of the objects to set landmarks on the back. In case you have problems defining landmarks, you may use an existing set of landmarks by loading the file `landmarkSet` from the directory `data/tutorials`. Once landmarks have been created, the next step is to transform the two objects into each other.

2.7.3 Registration via a Rigid Transformation

To register one object to the other, connect a *LandmarkWarp* module to the *Landmarks* object by clicking with the right mouse button on the *Landmarks* icon in the Pool and selecting *Compute LandmarkWarp*.

We want to perform an alignment of the first lobe to the second. Therefore the *LandmarkWarp* module must be connected to the image data of the first lobe (use the right mouse button in the white square of the *LandmarkWarp* icon).

The *LandmarkWarp* module is able to perform several transformations. We start with a purely rigid transformation to match the corresponding landmarks as well as possible by performing only rotations and translations of the first object. To do that choose

Method: Rigid

in the *LandmarkWarp* module and make sure that *Direction* is set to

Direction: 1 → 2.

Then press *Apply* to start the computation. The module creates a new data object named *lobus.Warped*. To visualize the result, connect the isosurface that was initially connected to the data of the first lobe to the result, select it and press the *Apply* button. In order to compare the result with the second lobe, adjust the viewer toggle of its *Isosurface* module to display it in the first viewer. You should see that the result of the transformation fits the second object quite well.

2.7.4 Warping Two Image Volumes

Using the rigid transformation the object will not be deformed. To perform a deformation and obtain a better fit we can use another transformation method of the *LandmarkWarp* module. Select the latter and choose

Method: Bookstein and press the *DoIt* button.

To visualize the result, the isosurface has to be recomputed. Having done that, you can see both the deformed and the second lobe in the first viewer. To merely see the resulting deformation in the first viewer, switch off the viewer toggle of the second lobe's *Isosurface* module. Only a little deformation will be seen because the two original objects were rather similar. Using more different data sets results in larger deformations.

2.8 Registration of 3D image datasets

In medical imaging a frequent task has become the registration of images from a subject taken with different imaging modalities, where the term *modalities* here refers to imaging techniques such as

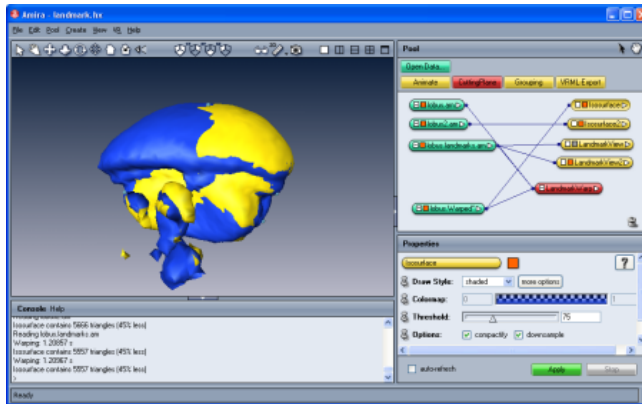


Figure 2.24: Result of landmark-based elastic warping using the Bookstein method.

Computed Tomography (CT), Magnetic Resonance Tomography (MRT) and Positron Emission Tomography (PET). The challenge in inter-modality registration lies in the fact that e.g in CT images "bright" regions are not necessarily bright regions in MRT images of the same subject.

In registration typically one of the datasets is taken as the *reference*, and the other one is transformed until both datasets match. *amira*'s *AffineRegistration* module provides an affine registration, i.e. it determines an optimal transformation with respect to translation, rotation, anisotrope scaling, and shearing.

Closely related to registration is the task of *image fusion*, i.e., the simultaneous visualization of two registered image datasets.

This tutorial shows how a registration can be performed and how to visualize the results. The following issues will be discussed:

1. Basic manual registration using the *Transform Editor*
2. Automatic registration
3. Image fusion

2.8.1 Basic Manual Registration

In this tutorial we want to register a CT and an MRT dataset of a patient, showing the pelvic region. The images are located in the *amira* data directory in the subdirectory *registration*.

- Load the files `data/registration/CT-data.am` and `data/registration/MRT-data.am` into *amira*.
- Attach an *OrthoSlice* module to each of the datasets.

- Select *Coronal* (or *xz*) at the *Orientation* port of the *OrthoSlice* module connected to the MRT data.
- Select a camera position for the 3D viewer where you can see both the axial slices of the CT data and the coronal slices of the MRT data.
- Select slice 31 at the *Slice Number* port of the *OrthoSlice* module connected to the CT data.

Now one *OrthoSlice* module should show an axial slice through the hip joints. Move the coronal slice through the MRT data. You will observe that the two datasets are not correctly aligned.

- Select the green icon of the MRT dataset. Invoke the *Transform Editor* by pressing the *Transform Editor* button in the *Properties Area*. The *Transform Editor* enables you to specify an affine transformation, including translation, rotation, and scaling. This transformation will be applied to the corresponding 3D dataset. You can edit the transformation interactively in the 3D viewer using different Open Inventor draggers. You can also enter transformations numerically.
- Press the *Dialog* button. A dialog window will pop up.
- Enter -2 at the third text field of the *Translation* port of the dialog window. This means a translation of -2 cm in the z direction.
- Enter 5 at the first text field at the *Rotation* port. This means a rotation of 5 degrees. The axis of rotation is defined at the next ports, here it is the z-axis.
- Press the *Close* button of the dialog window. Leave the *Transform Editor* by pressing again the *Transform Editor* button.

Inspect some coronal slices through the MRT dataset. Now there is a better alignment of the CT and MRT data, but it's still not perfect.

2.8.2 Automatic Registration

The *Registration* module provides an automatic registration via optimization of a quality function. For registration of datasets from different imaging modalities, the *Normalized Mutual Information* is the best suited quality function. In short, it favors an alignment which "maps similar gray value structures to similar gray value structures". A hierarchical strategy is applied, starting at a coarse resampling of the datasets, and proceeding to finer resolutions later on.

- Attach an *Registration* module to the MRT dataset by choosing *Compute/AffineRegistration* from the popup menu over the *MRT-data.am* icon.
- Connect the second input port *Reference* of module *Registration* to the CT dataset. For this click with the right mouse button on the white square at the left hand side of the module's icon.
- Select toggle *Extended options*. More ports of the *Registration* module will become visible.

The first three ports of the *Registration* module define the optimization strategy. The default settings mean that an *ExtensiveDirection* optimizer is used for the coarse levels and a *QuasiNewton* optimizer

for the finest two levels of the resampling hierarchy. At the *CoarsestResampling* port you can select the resampling rate for the coarsest resolution level. The default resampling rate is smaller in the z direction because the reference dataset has a finer resolution in the x and y direction (0.17 cm) than in the z direction (0.5 cm). For the default settings (8,8,3), the resampling hierarchy will consist of four levels: (8,8,3), (4,4,2), (2,2,1), and the original resolution, (1,1,1).

The *Normalized Mutual Information* is calculated from gray value histograms. The selected histogram ranges should enclose the essential information of each dataset. Normally you can choose the same range as for visualization via an *OrthoSlice* module.

- Set -200 and 200 at the two text fields of the *Histogram range reference* port.

At the *Transform* port you can specify the type of affine transformation. The default settings mean that only rigid body motions will be applied, i.e. translations and rotations.

Option *Ignore finest resolution* means that optimization is done on all but the finest level of the resampling hierarchy. This will slightly reduce the accuracy, but save a large amount of computing time.

Automatic registration may take some time depending on the resolution of the images and the quality of the pre-alignment. You can interrupt automatic registration at any time using the stop button. Interruption may take some seconds. The progress bar shows the current hierarchy level and the progress at that level.

- Start automatic registration by pressing the *Register* button at the *Action* port.

2.8.3 Image Fusion

The task of image fusion is the simultaneous visualization of two datasets. To that end *amira* offers for all types of slicing modules (*Orthoslice* and *ObliqueSlice*) the *Colorwash* module. Using *Colorwash*, the images from one dataset can be overlayed over that of another taking into account their orientation in space.

- Remove the *OrthoSlice* module connected to the MRT dataset.
- Select the green icon of the MRT dataset.
- Select a *Colorwash* module from the popup menu over the icon of the *OrthoSlice* module connected to the CT dataset.
- Select the yellow icon of the *Colorwash* module.
- Select the *physics.icol* colormap at port *Colormap*.
- Set 70 as the upper bound for the colormap range.
- Select the icon of the *OrthoSlice* module.
- Inspect axial slices with slice numbers between 15 and 45.

You will observe a good alignment of the pelvic bone from both datasets. The soft tissue contours are not perfectly aligned because there was some soft tissue deformation between both scans. This cannot be described by a rigid transformation.

In image fusion it is sometimes necessary to observe all three orthogonal directions simultaneously. For that the *StandardView* module can be used for image fusion. The *StandardView* module opens a separate window with four viewers, three of them showing the three orthogonal slices of the image data and a fourth being a new instance of the 3D viewer.

- Attach a *StandardView* module to the CT dataset by choosing *Display StandardView* from the popup menu over the *CT-data.am* icon. *amira's* *Viewer* window will now be split into four parts showing three orthogonal slices through the CT data, and the 3D Viewer in the upper left part.
- Connect the second input port *OverlayData* of the *StandardView* module to the MRT dataset. For this, click with the right mouse button on the white square at the left hand side of the module's icon.
- Select slice numbers 179, 149, and 31 at ports *Slice x*, *Slice y*, and *Slice z*, respectively. The three orthogonal slices will show the hip joints now.
- Increase the zoom-factor by clicking twice on button *>* at the *Zoom* port.
- Select *checkerboard* at the *Overlay mode* port.
- Vary the size of the checkerboard tiles by moving the slider at the *Pattern size* port. In this way you can again check the alignment of the CT and MRT datasets.

The bone contours around the hip joints show a good match. Note that bone is represented by white (i.e high intensity) voxels in the CT data, but may occur as both white and black voxels in the MRT data. In the axial slice you can observe larger deviations of the outer body contour between the CT and MRT data.

2.9 Alignment of 2D Physical Cross-subsections

Many microscopic techniques require the sample to be physically cut into slices. Then images are taken from each cross-subsection separately. Usually the images will be misaligned relative to each other. Before a 3-dimensional model of the sample can be reconstructed the images have to be aligned taking into account translation and rotation. This tutorial shows how this task can be performed using the *AlignSlices* module.

The following issues will be discussed:

1. Basic manual alignment
2. Alignment via landmarks
3. Optimizing the quality function
4. Resampling the input data

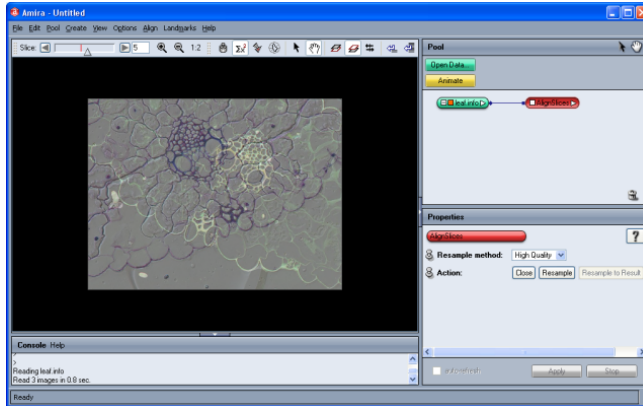


Figure 2.25: User-interface of the align tool.

5. Other alignment options

2.9.1 Basic Manual Alignment

In this tutorial we want to align 10 microscopic cross-sections of a leaf showing a stomatal pore. The images are located in the data directory in the subdirectory *align*. Each slice is stored as a separate JPEG image. The file *leaf.info* defines a 3D image stack consisting of the 10 individual slices. It is a simple ASCII file as described in the *stacked slices* file format subsection.

- Load the file `data/align/leaf.info`.
- Create an align module by choosing *Compute AlignSlices* from the popup menu over the *leaf.info* icon.
- Press the *Edit* button of *AlignSlices*.

A new graphics window is popped up allowing you to interactively align the slices of the 3D image stack. To facilitate this task, usually two consecutive slices are displayed simultaneously. One of the two slices is *editable*, i.e., it can be translated and rotated using the mouse. By default the upper slice is editable. This is indicated in the tool bar of the align window (the “upper slice” button is selected).

- If necessary, press the zoom out button (“-” magnifying glass) to allow the entire slice to be visible in viewer.
- Translate the upper slice by moving the mouse with the left mouse button pressed down.
- Rotate the upper slice by moving the mouse with the left mouse button and the Ctrl key pressed down. Alternatively, slices can be rotated using the middle mouse button.
- Make the lower slice editable by selecting the “lower slice” tool button. Translate and rotate the

lower slice.

- Hold down key number 1. While this key is hold down only the lower slice is displayed.
- Hold down key number 2. While this key is hold down only the upper slice is displayed.
- Pressing key number 1 and 2 also changes the editable slice. Note how the slice tool buttons change their state.

Other pairs of slices can be selected using the slider in the upper left part of the align window. Note that the number displayed in the text field at the right side of the slider always refers to the editable slice. The next or the previous pair of slices can also be selected using the space bar or using the backspace key, respectively. The cursors keys are used to translate the current slice by one pixel in each direction.

- Browse through all slices using the space bar and the backspace key. Translate and rotate some slices in an arbitrary way.
- Translate all slices at once by moving the mouse with the left mouse button and the Shift key pressed down.
- Rotate all slices at once by moving the mouse with the left mouse button and the Shift and Ctrl key pressed down.

Transforming all slices at once can be useful in order to move the region of interest into the center of the image.

2.9.2 Alignment Via Landmarks

Besides manual alignment, four automatic alignment options are supported, namely alignment using a principal axes transformation, automatic optimization of a quality function, edge detection-based alignment and alignment via user-defined landmarks. The principal axes method and the edge detection method are only suitable for images showing an object which clearly separates from the background. The optimization method requires that the images be already roughly aligned. Often such a pre-alignment can be achieved using the landmarks method.

Alignment via landmarks first requires you to interactively define the positions of the landmarks. This can be done in *landmark edit* mode.

- Activate *landmark edit* mode by pressing the arrow-shape tool button located between the hand-shape button and the edge detection button.

In *landmark edit* mode only one slice is displayed instead of two. Two default landmarks are defined in every slice.

- Click on one of the default landmarks. The landmark gets selected and is drawn with a red border.

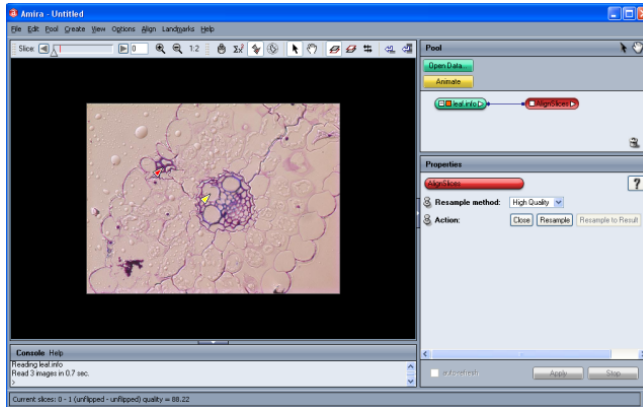


Figure 2.26: The figure shows how the landmarks should be set in the tutorial.

- Click somewhere into the image in order to reposition the selected landmark.
- Click somewhere into the image while no landmark is selected. This causes the next landmark to be selected automatically.
- Click at the same position again in order to reposition the next landmark.

The double-click method makes it very easy to define landmark positions. Of course, additional landmarks can be defined as well. Landmarks can also be deleted, but the minimum number of landmarks is two.

- Choose *Add* from the *Landmarks* menu.
- Click anywhere into the image in order to define the position of the new landmark.
- Select the yellow landmark by clicking on it.
- Choose *Remove* from the *Landmarks* menu in order to delete the selected landmark again.

Two landmarks should be visible now, a red one, and a blue one. Next, let us move these landmarks to some reasonable positions so that we can perform an alignment.

- Select slice number 0.
- Place the landmarks as shown in Figure 2.26. Make use of the double-click method.
- In all other slices place the landmarks at the same positions.

Once all landmarks have been set, we can align the slices. It is possible to align only the current pair of slices, or to align all slices at once. Note that all alignment actions as well as landmark movements can be undone by pressing Ctrl-Z.

- Switch back to *transform* mode by pressing the hand-shape tool button. Two slices should be displayed again.
- Align the current pair of slices by pressing the second tool button from the right (the one with only two lines).
- Align all slices by pressing the first tool button from the right (the one with many lines).
- Move and rotate the whole object into the center of the image using the mouse with the Shift key hold down.

In most slices the alignment now should be quite good. However, looking at the pairs 3-4 and 4-5 (displayed in the lower left corner of the align window) you'll notice that there is something wrong. In fact, slice number 4 has been accidentally inverted when taking the microscopic images. Fortunately, we can compensate for this error as follows:

- Select slice pair 3-4 and make sure that the upper slice, i.e., slice number 4, is editable.
- Invert the upper slice by pressing the invert button (third one from the right).
- Realign the current pair of slices by pressing the second button from the right).
- Select slice pair 4-5 and realign this pair of slices as well.

Alternatively, you could have aligned all slices from scratch by pressing the first button from the right.

2.9.3 Optimizing the Quality Function

Once all slices are roughly aligned, we can further improve the alignment using the automatic optimization method. At the bottom of the align window the quality of the current alignment is displayed. This is a number between 0 and 100, where 100 indicates a perfect match. The quality function is computed from the squared gray value differences of the two slices. The optimization method tries to maximize the quality function. Since only local maxima are found, it is required that the slices be reasonably well aligned in advance.

- Click on the slice in the viewer. The quality of the alignment is displayed in the status bar at the bottom of the window. Remember the current quality measure.
- Activate the optimization mode by pressing the tool button with the sum x squared symbol.
- Align the current pair of slices by pressing the second button from the right. Observe how the quality is improved.

Automatic alignment is an iterative process. It may take quite a long time depending on the resolution of the images and of the quality of the pre-alignment. You can interrupt automatic alignment at any time using the *Stop* button.

- Automatically align all slices by pressing the first button from the right.

2.9.4 Resampling the Input Data

If you are satisfied with the alignment, you can resample the input data set in order to create a new aligned 3D image. This is done using the *Resample* button of the *AlignSlices* module.

- Press the *Resample* button of the *AlignSlices* module.
- Attach an *OrthoSlice* module to the resulting object *leaf.align* and verify that the slices are aligned.

Sometimes you may want to improve an alignment later on. In this case it is a bad idea to align the resampled data set a second time, since this would require a second resampling operation. Instead, you could write the transformation data into the original image object and store this object in *AmiraMesh* format. After reloading the *AmiraMesh* file you can attach a new *AlignSlices* module and continue with the stored transformations.

- Choose *Save transformation* from the *Options* menu of the align tool. This will store the transformation data in the parameter subsection of the input object *leaf.info*.
- Delete the *AlignSlices* module.
- Save *leaf.info* in *AmiraMesh* format.
- Reload the saved object *leaf.am*.
- Attach a new *AlignSlices* module to *leaf.am* and click the *Edit* button. Note that the original alignment is restored.

2.9.5 Using a Reference Image

In some cases you might want to correct the alignment after image segmentation has been performed. In order to avoid segmenting the newly resampled image from scratch, you can apply the same transformations to a label field using a reference image.

- Delete any existing align tool.
- Load the file `data/align/leaf-unaligned.labels`.
- Attach a new *AlignSlices* module to the label field.

In the label field the guard cells of the stomatal pore are marked. Segmentation has been performed before the images were aligned. Now we want to apply the same transformation defined for the image data to the labels.

- Connect the *Reference* port of *AlignSlices* to *leaf.am* (this is done by activating the popup menu over the small white square at the left side of the *leaf.am* icon). Observe how the transformations are applied to the label field.
- Export an aligned label field by pressing the *Resample* button.

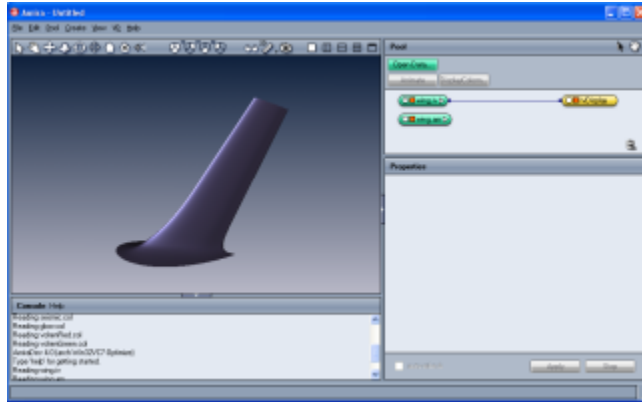


Figure 2.27: Open Inventor geometry of the airfoil.

The image volume used in this tutorial is an RGBA color field. However, the image segmentation editor only supports gray level images. Therefore you must convert the color field into a scalar field using *CastField* before you can invoke the image segmentation editor for the resampled labels.

2.10 Visualization of Vector Fields (Mesh Pack)

This step-by-step-tutorial briefly explains some *Mesh Pack* modules for vector field visualization. The use of these modules is explained by way of data representing the flow around an airfoil. The two methods referred to in steps 2 and 3 are independent of each other. These topics will be covered:

1. Loading the Wing and the Flow Field
2. Line Integral Convolution
3. Illuminated Stream Lines

2.10.1 Loading the Wing and the Flow Field

As in the previous tutorials, we will use the file dialog to import data.

- Import the geometry of the wing by loading the file `wing.iv` from the directory `data/tutorials`.
- Attach an *IvDisplay* module to this data object.
- Load the vector field data set called `wing.am` from the directory `data/tutorials`.

The extension `.iv` indicates that the wing geometry is defined in the Open Inventor file format. The *IvDisplay* module can be used for displaying geometry in this format. The vector field itself is stored

in *amira*'s native *AmiraMesh* file format. The data represents the flow around an airfoil computed on a regular grid with curvilinear coordinates. By selecting the green data icon *wing.am*, you can find out that the number of grid nodes in x,y,z-direction is 125 x 41 x 21.

2.10.2 Line Integral Convolution

Line Integral Convolution (LIC) is a method for visualizing 2D vector fields, i.e., depicting the vector field direction for a suitably sampled subset of points in the 2D domain. The direction is represented by local streamlines, i.e. by curves whose tangent vectors coincide with those of the given vector field. Local streamlines are computed such that all image pixels are covered. The streamlines are projected onto a random noise input texture map of the same size as the vector field domain. The projection step involves summations of texture pixel intensities along streamline paths by way of a convolution integral with a filter kernel. This causes pixel intensities in the resulting image to be highly correlated along individual streamlines but statistically independent perpendicular to the latter. Thus the directional structure of the vector field becomes clearly visible.

Here we use the 3D vector field that we have already loaded and visualize two-dimensional slices:

- Connect a *PlanarLIC* module to the vector field *wing.am* and select it.
- Choose the slice number 22 in the *Translate* port.
- Set filter length to 40 and resolution to 200.
- Press the *Apply* button.

Whenever the projection plane of the *PlanarLIC* module is changed or other values for filter length or resolution are taken, the LIC texture must be recalculated, i.e., the *Apply* button must be pressed again. Otherwise, a checkerboard pattern will be displayed. Experiment with different filter lengths and resolution values to see what kinds of textures can be produced.

To get an impression of the magnitude of the vectors, you can apply a colormap to the image. Some default colormaps are already loaded when *amira* starts up. The corresponding icons usually will be hidden. In order to use one of the colormaps, you have to select it explicitly:

- From the main window's *Pool / Show Object* menu select *temperature.icol* or any other colormap: the corresponding icon becomes visible.
- Select *Magnitude* from port *Colorize* of the *PlanarLIC* module. A new port labeled *Colormap* appears in the Properties Area.
- Connect the colormap port to *temperature.icol* by clicking with the right mouse button in the red rectangle and choosing *temperature.icol* from the appearing popup menu.

The two integer values of the colormap port specify the range of values to which the colormap is applied. Vector magnitudes within this range are depicted symbolically by coloring streamline pixels such that each pixel gets the unique color associated with the magnitude value by the *temperature.icol* colormap.

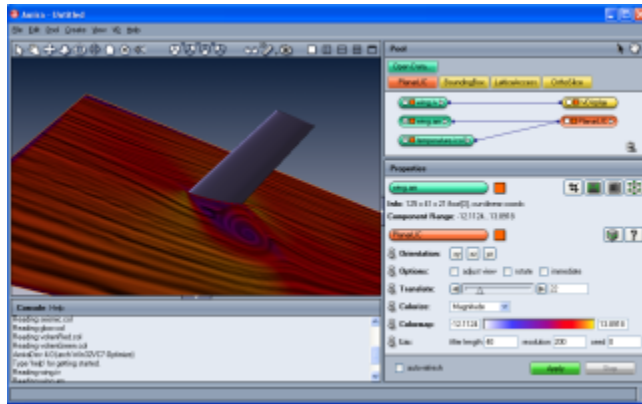


Figure 2.28: Air flow around the wing visualized using Line Integral Convolution

- Select the *wing.am* icon and read off the magnitude range.
- Shift-select the *PlanarLIC* icon and enter the magnitude range into the *Colormap* port.

2.10.3 Illuminated Stream Lines

Illuminated Stream Lines is a technique for interactive 3D vector field visualization which makes use of large numbers of properly illuminated stream lines. A realistic shading model is employed which significantly increases realism of the resulting images and enhances spatial perception.

Now you will learn which tools are used for illuminated stream line visualization and how to use them to get a 3D impression of our airflow vector field.

- Remove the *PlanarLIC* module or disable its display.
- Connect a *DisplayISL* module to the vector field *wing.am*.
- Set *Num Lines* to 300.
- Press the *Apply* button.
- Click on the *TabBox* button of port *Box* and zoom out the viewer if you cannot see a box.

A *TabBox* appears in the viewer. Only stream lines flowing through this box are visible. The green tabs at the corners and edges of the box allow you to change the dimensions of the box.

- Switch the viewer into interaction mode.
- Try out what happens if you click with the left mouse button on one of the green tabs on the corners or edges of the *TabBox* and drag them around.
- To move the whole *TabBox*, click with the left mouse button in the box and move it.

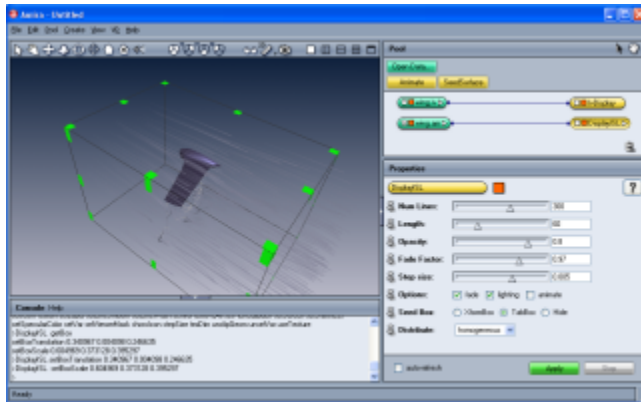


Figure 2.29: Illuminated streamlines around a wing

- Try to get the *TabBox* into a shape and position as shown in the image.
- Press *Apply* button again in order to recalculate the stream lines.

Now some information about Console Window commands. Most *amira* modules provide more control features than those that are available by the ports displayed in the Properties Area. All of them are available by commands that you type in the console window. You can get a list of commands associated with a particular module currently in use just by entering its name. Now we will use such commands to form and position the *TabBox* exactly as in the image above.

- Type “DisplayISL” into the Console Window.
- Type “DisplayISL getBox” into the Console Window.

The first command lists all commands of the *DisplayISL* module and the second shows the scale and translation of the current *TabBox*. The first word of a command must always be the name of a module as shown on its icon. Note that module commands are not recognized unless corresponding modules have been loaded into the Pool. However, you do not need to select a module for typing in its commands.

- Type “DisplayISL setBoxTranslation -0.31 0.00 0.17”
- Type “DisplayISL setBoxScale 0.11 0.04 0.14”

Now the sub-field that corresponds to the clipping of the vector field as implied by the settings of the *TabBox* should look like the one shown in the image above.

2.11 Creating animated demonstrations

In this tutorial you will learn how to use the *DemoMaker* module for creating an animated sequence of operations within *amira*. In our example, we will visualize a polygon model using effects such as transparency, camera rotation, and clipping to make the visualization more meaningful and attractive.

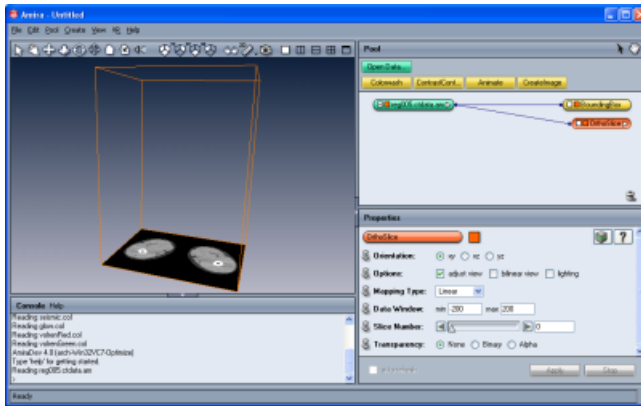
The tutorial covers the following topics:

- creating an initial network for the demo
- animating an *OrthoSlice* module
- activating additional modules during the demo
- using a camera rotation or path
- editing or removing events that are already defined
- overlaying a bone model with a transparent skin model
- using clipping to make the skin appear gradually over the bone
- advanced clipping issues
- inserting breaks and defining demo segments
- using function keys for jumping between demo segments
- defining partial loops within the demo sequence
- storing and replaying a demo sequence

Once you have learned how to define an animated demo sequence, you can further learn how to record the demonstration into a movie file in [Section 2.12](#).

2.11.1 Creating a Network

First, we need an *amira* network that contains all the data and modules for the visualization and animation we want to do. In our example, we pick the medical CT scan dataset `reg005`. Start by loading `data/medical/reg005.ctdata.am` from the *amira* root directory. By right-clicking on the green data icon and selecting from the dataset's popup menu, attach a *BoundingBox* module as well as a *OrthoSlice* module to the data. If you use the mouse to navigate around the model in the 3D viewer, you should manage to get a result similar to this:

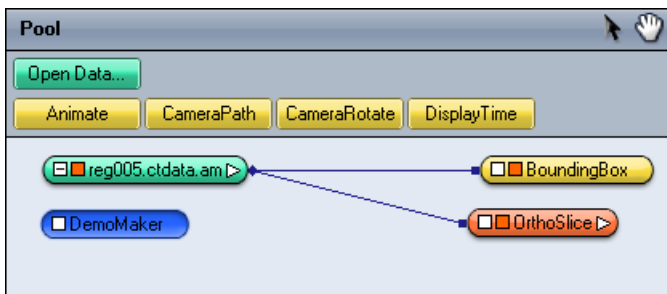


(load network)

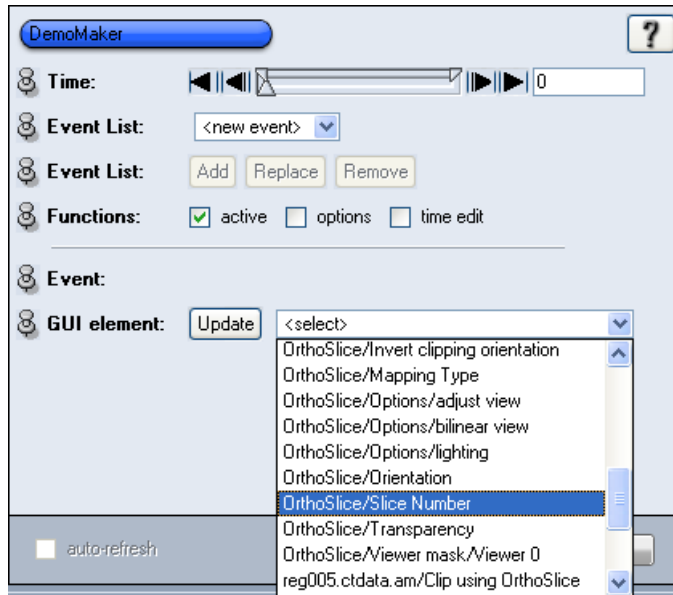
2.11.2 Animating an OrthoSlice module

Let us move the *OrthoSlice* plane up and down to show what the data looks like. Note that the *OrthoSlice* module has a port called *Slice Number*. If you change the value of that slider, you see the plane move in the viewer.

Now let us animate this slider using the *DemoMaker* module as our first exercise. From the menu bar, select *Create/Animation/Demo/DemoMaker*. A blue script object appears in the Pool:

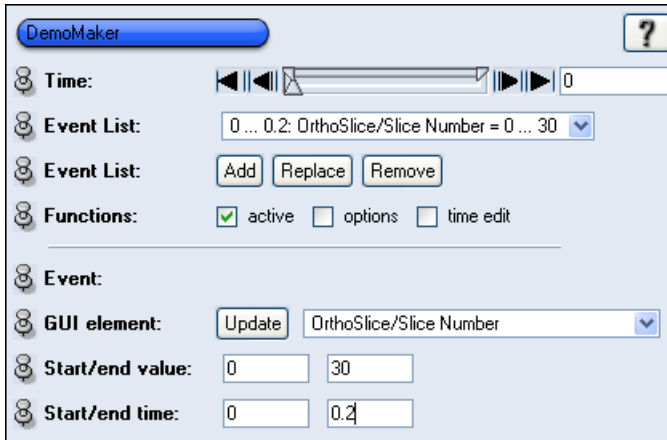


Click on the blue icon to see its user interface. Whenever you want to animate some port of the current network, you must select that port in the selection list called *GUI element*. Try to find the entry called *OrthoSlice/Slice Number*, which corresponds to the *Slice Number* port of the current *OrthoSlice* module. If you cannot find the entry, you may need to press the *Update* button to the left of the selection menu (see below for an explanation).



Once you have selected *OrthoSlice/Slice Number*, you see two more ports appear in the *DemoMaker* module: *Start/end value* as well as *Start/end time*. The start and end value specify between which two values the *OrthoSlice* slider will be moved. Click the mouse into the start or end value fields, hold down the *Shift* key, and drag the mouse with *Shift* held down. With this feature called the *virtual slider* you can quickly set the desired value within the allowed range. Set the start value to 0 and the end value to 30. Then, set the start time to 0 and the end time to 0.2. These time values specify times on the *Time* slider of the *DemoMaker* module (first port in the module). You have now specified an *event* starting at time 0 and ending at time 0.2, varying the *OrthoSlice* slice number between 0 and 30 during that time.

Now press the *Add* button in the *Event List* port to add the newly defined event to the list of events. This list of events is represented in the selection menu above the *Add* button.



Test the result by pressing the play button of DemoMaker's time slider, represented by the triangle pointing to the right. When you press it, the time slider will start moving from the left to the right. When the time value is between 0 and 0.2, you should see that the OrthoSlice plane is moving between the specified start and end values in the viewer (load network). You can also play your demo backwards using the play button to the left of the time slider, or simply click somewhere on the time slider to jump to any point in time of the demonstration.

If the demo sequence runs too slow or too fast, you can adjust this by right-clicking anywhere on the time slider and selecting *Configure* from the popup menu. Change the *Increment* value in the dialog box that appears. A smaller increment will make the animation slower, whereas a larger increment makes it faster. If you choose an increment value too large, the animation might become "jerky".

2.11.3 Activating a module in the viewer window

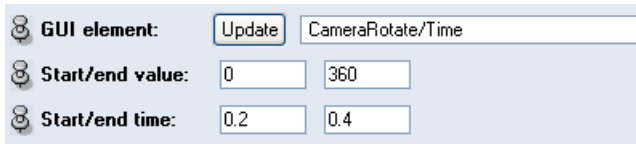
Next, let us add a visualization of the bone structure in the dataset after we have moved the OrthoSlice. Load the dataset `data/medical/reg005.surf` in addition to the current network. Attach a *SurfaceView* module to it. Click on the yellow *SurfaceView* module to see its user interface. Press the *Clear* button in the *Buffer* port, then select *Bone* and *All* in the *Materials* port and press the *Add* button in the buffer port. This will visualize the bone surface of the model.

2.11.4 Using a camera rotation

To look at the 3D patient model from all sides, let us add a camera rotation to our demo sequence. Select *Create/CameraRotate* from the menu. Try the rotation by playing the time slider in the *CameraRotate* module. If you do not like the axis of rotation, reset the time slider to 0, navigate to a good starting view in the viewer window, and click on *recompute* in the *CameraRotate* module. Note that the values of the *CameraRotate* time slider range from 0 to 360.

Once you are satisfied with the camera rotation, add it to the event list:

- Click on the *DemoMaker* module.
- Click on the *Update* button.
- Select *CameraRotate / Time* from the *GUI element* list.
- Enter 0 and 360 as the start and end value.
- Enter 0.2 and 0.4 as the start and end time.
- Click on the *Add* button in the *Event List* port.



The screenshot shows a GUI window with three rows of controls. The first row is labeled 'GUI element:' and contains an 'Update' button and a text box with 'CameraRotate/Time'. The second row is labeled 'Start/end value:' and contains two text boxes with '0' and '360'. The third row is labeled 'Start/end time:' and contains two text boxes with '0.2' and '0.4'.

Now play the demo to see the result. After moving the slice and switching on the bone model, the view is rotated so that the bone can be seen from all sides (load network).

2.11.5 Editing or removing an already defined event

When you look at the demo sequence so far, you may think that it would be nice to wait for a short time before rotating the bone model. This can be done by starting the rotation at a later time step. We can easily correct this in the *DemoMaker* module:

- Select the *0.2 ... 0.4: CameraRotate* event in the *EventList* port.
- You see the start/end value and start/end time of this event appear in the lower part of the *DemoMaker* module.
- Change the start/end time to 0.3 and 0.5.
- Click on the *Replace* button in the *Event List* port. This replaces the currently selected event in the list by the event as defined in the lower part of the module.

Now you have moved the camera rotation event from 0.2-0.4 to 0.3-0.5 on the time line. Check the results by playing the time slider (load network).

Please note that you can delete an event from the list by simply selecting it from the *Event List* menu and clicking the *Remove* button.

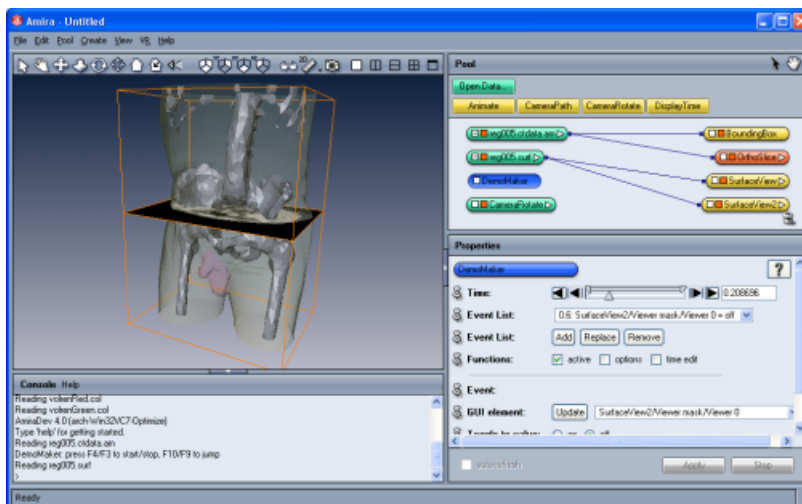
2.11.6 Overlaying the bone with skin

Now we want to show the patient's outer surface overlayed over the bone model.

- Attach a second *SurfaceView* module to the *reg005.surf* dataset. Since *Exterior* and *All* are selected as the default materials, this brings up the patient's exterior surface.
- Click on the second *SurfaceView* module. It should be called *SurfaceView2*.
- Select *transparent* from the *Draw Style* port.
- It will be helpful to show the bone underneath the exterior surface, so jump to time step 0.2 or later in the *DemoMaker* module.
- Adjust the grade of transparency using the *BaseTrans* slider in *SurfaceView2*.
- Smooth out the outer surface by clicking on *more options* in the *Draw Style* port and selecting *Vertex normals*.

Like we did with the bone model, we can switch on the skin model at some point in the demo sequence:

- Click on the *DemoMaker* module.
- Click on the *Update* button in the *GUI element* port.
- Select *SurfaceView2/Viewer Mask/Viewer 0* from the *GUI element* list.
- Check *on* in the *Toggle to value* port.
- Enter *0.6* in the *Trigger time* port.
- Click on *Add* in the *Event List* port.



Again, check out the results by playing the demo sequence.

2.11.7 Using clipping to add the skin gradually

Instead of just switching the skin on at one point, we can make it appear gradually over the bone from bottom to top. In order to do so, we use the *OrthoSlice* plane to *clip* the skin model, and then move the *OrthoSlice* plane up.

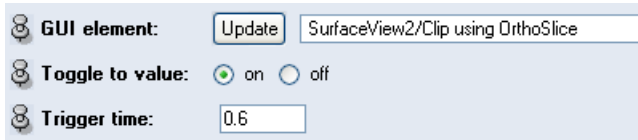
First, we need to move the *OrthoSlice* plane down again to where we want to start the clipping:

- Click on the *DemoMaker* module.
- Select *OrthoSlice/Slice Number* from the *GUI element* list.
- Enter 30 and 3 as the start/end values.
- Enter 0.3 and 0.5 as the start/end time.
- Click on *Add* in the *Event List* port.

Now, when you play the demo, the *OrthoSlice* plane will move down again during the camera rotation (load network).

Now, we will clip the skin model using the *OrthoSlice* plane:

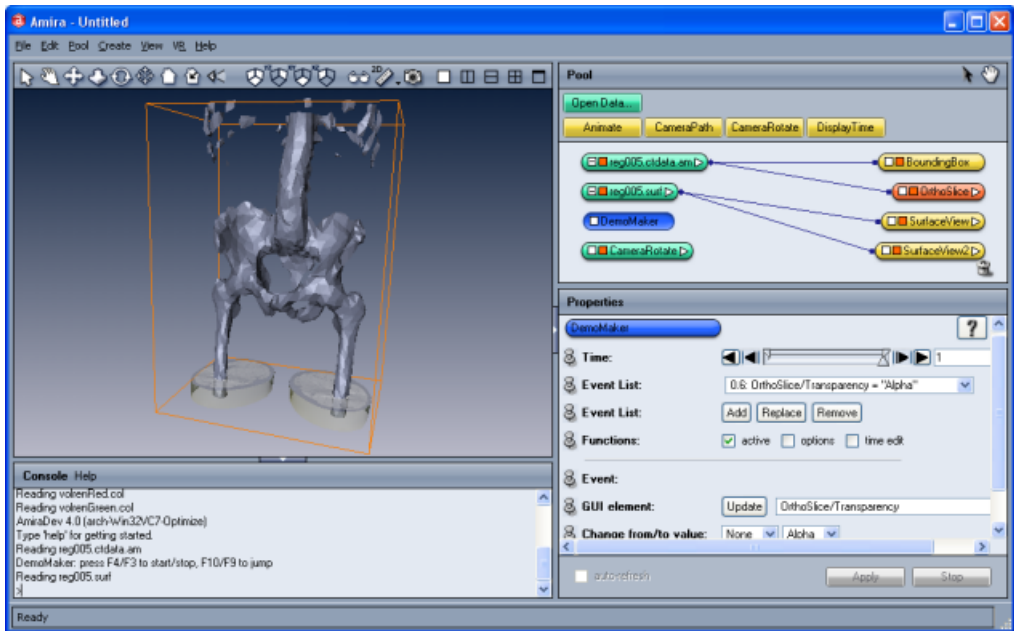
- Click on the *DemoMaker* module.
- Select *SurfaceView2 / Clip using OrthoSlice* from the *GUI element* list.
- Enter *on* as toggle value and *0.6* as trigger time.
- Click on *Add* in the *Event List* port.



When you run the animation now, you will not see the skin surface. This is because it is clipped above the *OrthoSlice* plane, and only visible below that plane. To see the partial surface below the plane, we must make the *Orthoslice* display transparent:

- Click on the *DemoMaker* module.
- Select *OrthoSlice / Transparency* from the *GUI element* list.
- Select *None* and *Alpha* as the from/to values.
- Enter *0.6* as the trigger time.
- Click on *Add* in the *Event List* port.

This way we have specified that at time 0.6, the *Transparency* port of the *OrthoSlice* module will be changed from value *None* to the new value *Alpha*. When running the demo sequence, the result should look like this:



As you see, part of the skin model is showing below the transparent *OrthoSlice* plane. To show all of the skin, we simply move the plane upwards pretty much the same way we did before:

- Click on the *DemoMaker* module.
- Select *OrthoSlice / Slice Number* from the *GUI element* list.
- Enter 3 and 58 as the start/end value.
- Enter 0.6 and 0.9 as the start/end time.
- Click on *Add* in the *Event List* port.

GUI element:	Update	OrthoSlice/Slice Number
Start/end value:	3	58
Start/end time:	0.6	0.9

Now you see the skin slowly appearing over the bone as the clipping plane moves upwards.

As a last step, you might want to rotate the view again while the skin is appearing. You can simply reuse the old camera rotation during a second time range:

- Click on the *DemoMaker* module.
- Select *0.3 ... 0.5: CameraRotate* from the *Event List* menu.

- You will see start/end value and start/end time appear in the lower part.
- Change the start/end time to 0.6 and 0.9.
- Click on *Add* in the *Event List* port. This will leave the old event untouched and add a second camera rotation event to the list.

GUI element: CameraRotate/Time

Start/end value:

Start/end time:

You can check out the final animation by loading a saved network script.

2.11.8 More comments on clipping

Clipping can sometimes be a little bit more complicated than in our example, because clipping can be applied to a plane in two different orientations. This means that you can either clip away everything *above* the plane, or *below* the plane. Unfortunately it is not always obvious which of the two cases you are in.

However, you can simply invert the orientation of the clipping in *DemoMaker*. In our example, you would simply select *OrthoSlice / Invert clipping orientation* from the *GUI element* port and add that event at the very beginning of your demo sequence (e.g., at some time before the clipping takes effect).

GUI element: CameraRotate/Time

Start/end value:

Start/end time:

☐ auto-refresh

OrthoSlice/Colormap/max
OrthoSlice/Colormap/min
OrthoSlice/Contrast Limit/
OrthoSlice/Data Window/max
OrthoSlice/Data Window/min
OrthoSlice/Frame
OrthoSlice/Invert clipping orientation
OrthoSlice/Mapping Type
OrthoSlice/Options/adjust view

You do not need to use an *OrthoSlice* module to do clipping. As you have seen, the *OrthoSlice* might occlude parts of what you want to show. In that case, it is better to create an empty *ClippingPlane* module by selecting *Create/Clipping Plane* from the menu. Attach the module to the dataset you want to clip (e.g., to *reg005.surf* in our example), and then use the *ClippingPlane* for clipping just as you used the *OrthoSlice* before.

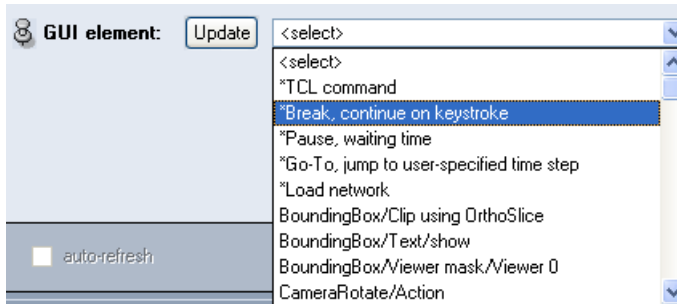
2.11.9 Breaks and Function Keys

The demo sequence that we have created in this tutorial automatically runs through the complete time range that we defined. Sometimes it might be desirable to split the sequence into several segments, so

that the demo will stop at some point and can be continued whenever the user desires to do so.

To take this into account, you can insert *breaks* in the *DemoMaker* event list. Let us insert one such break right after the bone model appears:

- Click on the *DemoMaker* module.
- Select **Break, continue on keystroke* from the *GUI element* list.
- Enter 0.21 as the trigger time.
- Click on *Add* in the *Event List* port.



This way the demo will stop at time 0.21, which is right after the time when the bone model is switched on (0.2). When you play the demo from the start, you will notice that after the bone is switched on, the demo will stop.

Let us insert a second break at time step 0.51, which is right before the skin is starting to show. Proceed as above, using a trigger time of 0.51 instead of 0.21 (load network).

If you run the demo from the very beginning, it will stop after the bone is displayed, and you can read a message in the console window telling you that *DemoMaker* just stopped and you may press F4 to continue. Try this by pressing the function key F4. The demo continues.

Likewise, the demo will stop just before showing the skin. Again, you can continue the demo by pressing F4. In general, at any point while the demo is running, you can press the F3 key to stop it manually. Pressing F4 will continue from the point where the demo stopped.

If you have defined breaks as we did above, there are two more interesting function keys that in some sense allow you to *navigate* through the demo segments: pressing F9 will jump back to the previous break or to the very beginning of the demo, and F10 will jump to the next break, or to the very end of the demo. If you use F9 / F10 when the demo is stopped, it will just jump, and you need to press F4 to start playing it from the new time step. If you press F9 / F10 while the demo is running, it will just jump to the new time step and continue running.

Please note that you can disable the breaks by checking the *skip break* toggle in the *Options* port of the *DemoMaker* module. You may even disable the definition of function keys by checking the *options* toggle in the *Functions* port, and then unchecking *function keys* in the second *Options* port. This is

especially important if you want to use multiple *DemoMaker* modules, since only one of the modules can define the keys.

Options: ☐ skip break ☐ skip pause ☒ skip load

Options: ☐ auto start ☐ function keys

Options: ☒ explicit redraw ☐ debug ☐ wait screen

2.11.10 Loops and go-to

One more feature that might be required for certain kinds of demos is the definition of loops. If you just want the whole demo to run in a loop, you can do this easily using the built-in features of the *time slider*: right-click on the slider and select *loop* or *swing*. Now if you play the time slider, it will start over from the beginning (loop mode), or play forwards, backwards, forwards... (swing mode).

However, you may want to define some part of the demo to run in a loop, and then stop the loop and continue with the demo upon key press. You can easily do this with the *go-to* feature of *DemoMaker*:

- Click on the *DemoMaker* module.
- Select **Go-to, jump to user-specified time step* from the *GUI element* list.
- Enter 0.19 as the *Trigger time*.
- Enter 0.0 as the *Time to jump to*.
- Click on *Add* in the *Event List* port.

GUI element: *Go-To, jump to user-specified time step

Trigger time:

Time to jump to:

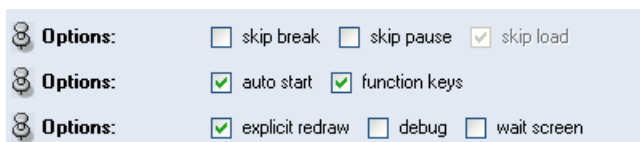
When you run the demo sequence now, it will loop in the first segment, only showing the *OrthoSlice* move up, jump down, move up again... You can stop this by clicking on the stop button of the *DemoMaker* time slider or by pressing F3. To continue after the loop, you need to jump to the next segment by pressing F10, and then start playing again by pressing F4.

2.11.11 Storing and replaying the animation sequence

As you may have noticed by now, storing a demo sequence once you have defined it is quite easy: simply save the whole *amira* network by selecting *File / Save Network...* from the menu. The *DemoMaker* module will be saved along with the network, and so will the demo sequence you have defined.

When you load the network back into **amira**, the state of the network will be the same as it was when you saved it. This means that you should be careful to reset the *DemoMaker* time slider to 0 before saving the network, if you want the demo to start from the beginning.

After loading the network, you can start the demo by clicking on the play button of the *DemoMaker* module, or by pressing F4. If you want to run the demo automatically right after the network is loaded, you can use the *auto start* feature that you find when you check *options* in the *Functions* port:



Just check the *auto start* toggle and save the network. When you load it again, the demo will start running automatically (load network).

2.12 Creating movie files

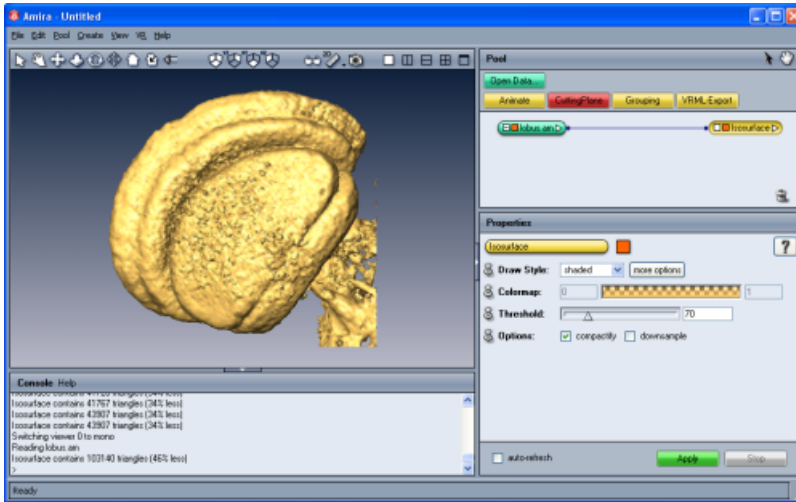
In this tutorial you will learn how to record a self-created animated sequence into a movie file using the *MovieMaker* module.

In our first example we will just use a camera path to animate the scene, whereas in our second example we will rely on the demo sequence created in [Section 2.11](#).

2.12.1 Attaching MovieMaker to a camera path

If you have created a visualization of your data and want to create a movie showing this visualization from all sides or from certain interesting viewpoints, you can create an appropriate camera path and record a movie by following the camera along that path.

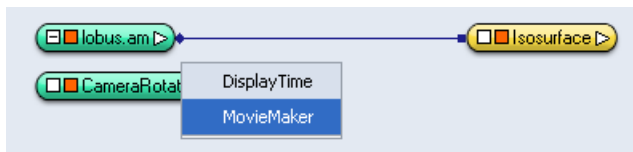
Let us create a simple example. Load the `lobus.am` dataset from the `tutorial` subdirectory and attach an *Isosurface* module to it. Choose an isosurface threshold of 70 and press the *Apply* button. The result should look similar to this:



The easiest way to create a simple camera path is to use the *CameraRotate* module. Select *Create/CameraRotate* from the menu, and press the play button of the newly created module. You can watch the scene rotate in the viewer while the time slider is playing (load network).

To record an animated scene into a movie file, you need to attach a *MovieMaker* module to a module that possesses a *time slider port*. The movie is recorded by going through the individual time steps and taking snapshots of the viewer along the way.

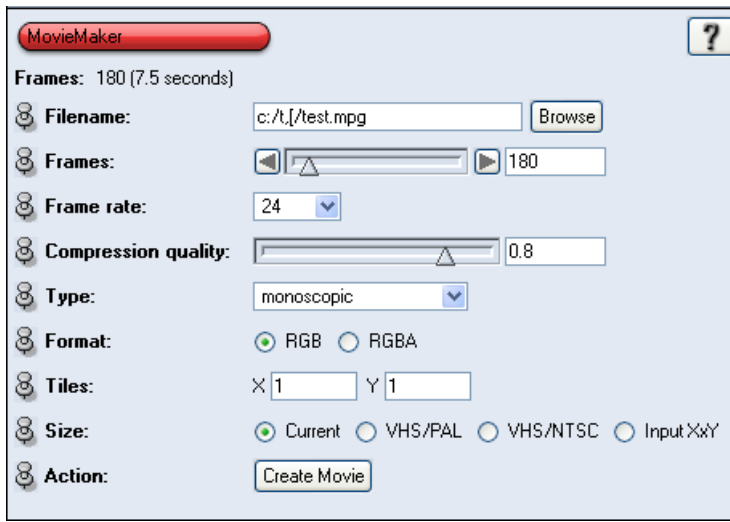
In our example, the *CameraRotate* module has a time slider, so we can attach a *MovieMaker* module to it by right-clicking on the *CameraRotate* icon in the Pool and selecting *MovieMaker* from the popup menu:



In the *MovieMaker* module, first click on the *Browse* button in the *Filename* port and enter a movie file name like `c:/tmp/test.mpg`. The `.mpg` suffix suggests that the movie file format will be MPEG, which is a widely accepted standard format for digital movies achieving a good compression ratio.

Next, adjust the parameters of the *MovieMaker* module to your liking, e.g., change the *number of frames*, the *image size*, or the *compression quality*. Please refer to the *MovieMaker* documentation for details.

In our example, let us choose 180 frames and leave all other parameters untouched. Since the *CameraRotate* module does a full rotation of 360 degrees, each of the 180 frames will represent a rotation of two degrees with respect to the previous frame. Press the *Create Movie* button to start recording.



Wait for some time while the *MovieMaker* module drives the *CameraRotate* module and accumulates the snapshots. *Please note that the speed during the recording process is different than the playback speed of the movie.* Now view the resulting movie file `test.mpg` with a movie player of your choice (e.g., Windows Media Player or a similar tool). Experiment with the recording parameters until you get the desired result (e.g., control the file size and image quality by changing the *Compression quality* value, choose different image sizes to see up to which image size your computer is capable of smoothly displaying the movie, and change the number of frames to control the speed of the rotation).

2.12.2 Attaching MovieMaker to DemoMaker

Now we try to record a movie of a more complex animated scene. To this end, we load one of the networks that we have created in in Section 2.11: load network.

As you might remember, the basic idea of the *DemoMaker* module was that you define a set of events to be executed on a certain time line. Check this out by clicking the play button of the time slider in the *DemoMaker* module. You should see a nicely animated demonstration.

If you remember the previous section in this tutorial, you might already have an idea of how we can record this animated demonstration into a movie file. Like the *CameraRotate* module in the first example, the *DemoMaker* module is controlled via a *time slider* that we can attach to. So simply right-click on the *DemoMaker* icon in the Pool and attach a *MovieMaker* module. Like before, enter a movie file name and select the number of frames before you click on the *Create Movie* button to start recording.

2.13 Using MATLAB Scripts

In this tutorial you will learn how to integrate complex calculus into **amira** using MATLAB (The MathWorks, Inc) by the means of the *CalculusMatlab* module.

In order to use the CalculusMatlab module, MATLAB 7 must be correctly installed on your computer. The CalculusMatlab module establishes a connection to the MATLAB computational engine that was registered during installation. If you did not register during installation, on the Windows command line you can enter the command:

- `matlab /regserver`

The limitations of the CalculusMatlab module are listed in its *documentation*.

This tutorial covers the following topics:

1. Loading and executing a MATLAB script.
2. Lowpass filtering on images.
3. Controlling the script with a time slider.
4. Thresholding on a volume.

2.13.1 Lowpass filtering on images

In this section we will learn how to apply a lowpass filter on an **amira** image using the MATLAB Fourier transformation. This example shows how to pass data and control variables from **amira** to MATLAB, execute a MATLAB script, and import the data back into **amira**.

- Load the *lena.png* image file located in subdirectory *data/tutorials/matlab*.
- Choose *Luminance* in the *Channel Conversion field* as shown in Figure 2.30.
- Right click on the green icon and choose CalculusMatlab from the Compute section.

A new red icon appears, the CalculusMatlab module that will try to connect to the MATLAB engine. This may take a while.

- Load the script *lowpass.m* located in subdirectory *data/tutorials/matlab* by clicking the *Read* button of the *File* port.
- Execute the script by clicking on the *buffer* button of the *Execute* port.
- Connect an *OrthoSlice* to the filtered image *result*.

The module uses the MATLAB computation engine which has its own user interface.

You can easily show or hide the MATLAB console using the checkbox in the options field. The MATLAB console is very useful for debugging purposes because it allows you to access variables of the MATLAB workspace. Any variable not cleared by the MATLAB "clear" command in the script is

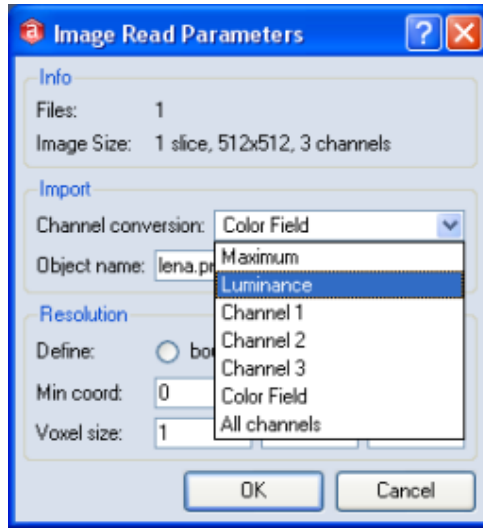


Figure 2.30: Loading the image

accessible in the MATLAB workspace, even after finishing the current CalculusMatlab computation (see the *CalculusMatlab* documentation).

In addition you can control scalar parameters of the script using time sliders:

- Create a time slider (*File/Create/Data/Time*).
- Connect the CalculusMatlab module to the time slider.
- Change the line `cutoff=0.05` to `cutoff=t` in the script (see the *CalculusMatlab* documentation for more information about the keyword `t`).
- Click on the time slider and adjust the value that will be assigned to `w`. Right mouse click in the text field of the time slider and select *Configure* to adjust the data range of the parameters.

Note: To handle RGBA image filtering, you must load the image with Color Field Channel Conversion and treat each channel separately in the script.

2.13.2 Thresholding on a volume

In this section we will learn how to apply a threshold to a volume. This is done by setting a value for a threshold. If the value for the voxel is less than the threshold, the voxel value is assigned the value of zero. If it is above the threshold, it is assigned a value of 255.

- Load the file *lobus.am* located in subdirectory *data/tutorials*.

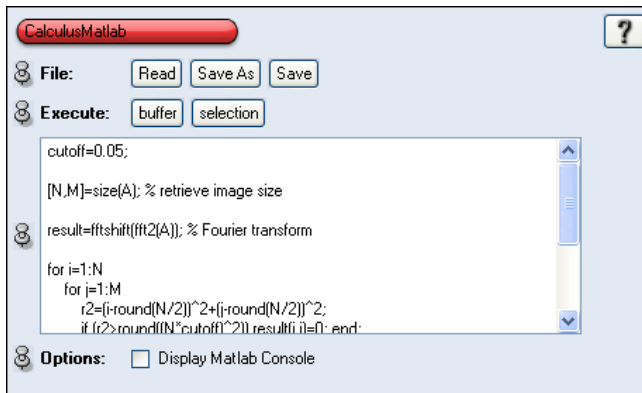


Figure 2.31: The CalculusMatlab module



Figure 2.32: Left: original Right: lowpass filtered

- Right click on the green icon and choose *CalculusMatlab* from the *Compute* section.
- Load the script *threshold.m* located in subdirectory *data/tutorials/matlab* by clicking the *Read* button of the *File* port.
- Execute the script by clicking on the *buffer* button of the *Execute* port.

A new green icon appears, the Lattice that will hold the threshold result. Connect an *OrthoSlice* or a *Voltex* to see the result.

Note: Any variable accessed by MATLAB and pushed back into the *amira* workspace will lose its voxel size information. You will need to correct the voxel size manually using the *Crop Editor*.

Chapter 3

Program Description

This chapter contains a detailed description of **amira** interface components and data types. No in-depth knowledge of **amira** is required to understand the following sections, but it is a good idea to have a look at one of the tutorials contained in Chapter 2, particularly the very first one described in Section 2.1 (Getting Started).

3.1 Interface Components

In this section the following interface components are described:

- *File Menu, Edit Menu, Pool Menu, Create Menu, View Menu, Help Menu*
- *Main Window, Viewer Window, Console Window*
- *File Dialog, Job Dialog, Preferences Dialog, Snapshot Dialog, System Information*

3.1.1 File Menu

The file menu lets you load and save data objects as well as **amira** network scripts. In addition, it gives you access to **amira**'s job dialog and allows you to quit the program. In the following text, all menu entries are discussed separately.

3.1.1.1 Open Data

The *Open Data* button activates **amira**'s *file dialog* and lets you import data sets stored in a file. Most file formats supported by **amira** will be recognized automatically via the file header or the file name extension. For each file, the file dialog will display its format. If you try to load a file for which the format couldn't be detected automatically, an additional dialog pops up asking you to select the format

manually. You may also manually set the file format for any file by selecting the file, activating the file dialog's popup menu using the right mouse button, and then choosing the *Format* option.

A list of all *supported file formats* is contained in the reference manual. Hints on how to import your own data sets are given in Section 4.1.

If you select multiple files in the file dialog, all of them will be loaded, provided all of them are stored in the same format. 2D images stored in separate files usually will be combined into a single 3D data object. On the other hand, there are some file formats which cause multiple data objects to be created. Finally, you can also import and execute **amira** network scripts using the *Load* button.

3.1.1.2 Open Time Series Data

This button also activates the *file dialog*, but in contrast to the ordinary *Load* option it is assumed that all selected files represent different time steps of a single data object. When loading such a time series an instance of a *time series control* module is created. This module provides a time slider allowing you to adjust the current time step. Whenever a new time step is selected the corresponding data file is read, and data objects associated with a previous time step are replaced. The module also provides a cache so that the data files only need to be read once provided the cache is large enough.

3.1.1.3 Save Data

The *Save Data* button allows you to save a single modified data object again using the same filename previously chosen under *Save Data As*. The button will only be active if the data object to be saved is selected and if this data object already has been saved using *Save Data As*. A common application of the *Save* button is to store intermediate results during manual segmentation in **amira**'s *image segmentation editor*.

3.1.1.4 Save Data As

This button lets you write a data object into a file. To do so you must first select the data object (click on the corresponding green data icon). Then choose *Save Data As* to activate **amira**'s *file dialog*. The file dialog presents a list of all formats suitable for saving that data object. Choose the one you like and press *OK*. Note that you must specify the complete file name including the suffix. **amira** will not automatically add a suffix to the file name. However, it will update the suffix whenever you select a new format from the file format list. Also, **amira** will ask you before it overwrites an existing file.

Some file formats create multiple files for a single data object. For example, each slice of a 3D image data set might be saved as a separate raster file. In this case, the file name may contain a sequence of hashmarks. This sequence will be replaced by consecutive numbers formatted with leading zeros.

If no file format at all has been registered for a certain type of data object, the *Save as* button will be disabled. It will also be disabled if more than one data object is selected in the Pool.

3.1.1.5 New Network

This button does a *Remove all objects* so that you can start building a new network in the Pool. It also sets the new network name to Untitled.hx.

3.1.1.6 Open Network

The *Open Network* button activates *amira*'s *file dialog* and lets you load a network stored in a file. Network files show up in the dialog as being of format "Amira Script". A *Remove all objects* will be done before the new network is loaded so that effectively the new network replaces the old network in the Pool. Currently only files with extension .hx can be opened.

3.1.1.7 Save Network

This button allows you to save the complete network of icons and connections shown in the Pool. If the network has not been previously saved, you will need to specify the name of an *amira* network script in the file dialog. When executed, the network script restores all data objects and modules as well as the current object transformations and the camera settings. The feature is useful for resuming work at a point where it was left in a previous *amira* run.

Note that usually all data objects must have been stored in a file in order to be able to save the network. If this is not the case, a dialog is popped up listing all the data objects that still need to be saved. In the dialog you can specify that all required data objects should be saved automatically in a separate subdirectory.

Instead of the option *amira script* you can also choose *amira script and data files (pack & go)* from the file dialog's format menu. In this case *all* data objects currently loaded will be saved in a separate directory. More options affecting the export of network scripts can be adjusted in the *Preferences* dialog.

3.1.1.8 Save Network As

This button works like the *Save Network* button except that in this case you will always need to specify the name of an *amira* network script in the file dialog.

3.1.1.9 Recent Files

This button can be used to load recently used files. When choosing this menu entry a submenu appears listing the five most recent files. If multiple 2D images have been loaded this is indicated with the name of the first file followed by three periods (...).

3.1.1.10 Recent Networks

This button can be used to load recently used network scripts. When choosing this menu entry a submenu appears listing the five most recent network scripts.

3.1.1.11 Jobs

This button brings up *amira*'s *job dialog* which is used to control the execution of batch jobs running in the background. For example, tetrahedral grids can be generated in a batch job (see module *TetraGen*). However, for most users the batch queue will be of minor interest.

3.1.1.12 Quit

This button terminates *amira*. The current network configuration will be lost unless you explicitly save it using *Save Network*.

3.1.2 Edit Menu

The *Edit* menu provides access to the standard cut/copy/paste/delete commands, as well as to an extended version of the Parameter Editor and the *amira* preferences dialog.

3.1.2.1 Cut

In the *Console*, this command cuts selected text and copies it to the clipboard. In the *Pool*, this command removes the selected objects.

3.1.2.2 Copy

In the *Console*, this command copies the selected text to the clipboard. In the *Pool*, this command has no effect.

3.1.2.3 Paste

In the *Console*, this command pastes the text in the clipboard to the current text insertion point. In the *Pool*, this command has no effect.

3.1.2.4 Delete

In the *Console*, this command deletes the selected text. In the *Pool*, this command deletes the selected objects.

3.1.2.5 Select All

In the *Pool*, this command select all objects.

3.1.2.6 Database

The *Database* button activates an extended version of the *Parameter Editor*, allowing you to manipulate *amira*'s global parameter database. Among others, the parameter database contains a set of predefined materials (to be used for image segmentation and surface reconstruction) and of predefined boundary ids (to be used for surface editing and FEM pre-processing). For example, for each material and for each boundary id a default color can be defined in the database.

Modification, insertion, and removal of parameters is performed in the same way as in the ordinary parameter editor. In addition, the extended parameter dialog provides a menu bar allowing you to load, import, save, or search the global parameter database. *amira*'s default database is stored in the file `share/materials/database.hm` located in the directory where *amira* was installed. Use the *Database* menu option *Set default file* to specify that a different database file be used instead. This change is permanent, i.e., it takes effect also if *amira* is restarted. To switch back to the system default, use the *Database* menu option *Use system default file*.

3.1.2.7 Preferences

This option opens the *amira* preferences dialog described in Section 3.1.12. The dialog controls how network scripts are exported. It also lets you choose if warning dialogs should be popped up if you try to delete data objects which have not yet been saved to file, or if you try to exit *amira* without having saved the current network before. Possibly most interesting, it allows you to configure the layout of your *amira* window.

3.1.3 Pool Menu

The *Pool* menu provides control over the visibility of object icons and lets you delete or duplicate objects. Depending on how many icons are selected in the *Pool*, some menu options might be disabled.

3.1.3.1 Hide

The *Hide* button hides all currently selected objects. The object's icons are removed from the *Pool* but the objects themselves are retained. You get the same effect by pressing the `Ctrl-H` key. Hidden objects can be made visible again using *Show* or *Show All*.

3.1.3.2 Remove

The *Remove* button deletes all selected objects and removes the corresponding icons from the *Pool*. You can get the same effect by pressing `Ctrl-X`. If you want to reuse a data object later on, be sure to

save it in a file before deleting it. If a data object has been modified but has not yet been saved to a file, it is marked by a little asterisk in the object icon. In the Preferences dialog you can choose whether a warning dialog should be printed if you try to delete unsaved data objects which cannot be recomputed by an up-stream compute module. If you delete a data object, all connected modules will be deleted as well. However, if you delete a module connected data objects (e.g., the results of a compute module) will be retained.

3.1.3.3 Duplicate

The *Duplicate* button creates copies of all selected data objects. For each copy a new data icon is put in the Pool. The name of a duplicated data object differs from the original one by one or more appended digits. The duplicate option is not available if you have selected icons that do not represent data objects (e.g., display or compute modules).

3.1.3.4 Rename

This button allows you to change the name of a selected object in a small dialog box which is popped up when the button is pressed. If no object is selected or if multiple objects are selected the button is disabled. Note that no two objects in **amira** can have the same name. Therefore, the name entered in the dialog may be modified by appending digits to it, if necessary.

3.1.3.5 Show

The *Show* button allows you to make hidden objects visible, so that their icons are displayed in the Pool. Among the hidden objects there are usually some colormaps which are loaded at start-up. This option will be unavailable if there are no hidden objects.

3.1.3.6 Show All

The *Show All* button makes all currently hidden objects visible, so that their icons are displayed in the Pool. This option will be unavailable if there are no hidden objects.

3.1.3.7 Remove All

The *Remove All* button deletes all currently visible icons and the associated objects from the Pool. A pre-loaded colormap that is currently visible is also deleted, but all hidden objects are retained. If you select the option *check if data objects need to be saved* in the Preferences dialog, a warning dialog is popped up if there are data objects which have not yet been saved to a file.

3.1.4 Create Menu

The *Create* menu lets you create modules or data objects that cannot be accessed via the popup menu of any other object. The *Create* menu provides different categories like the popup menu in the Pool. For example, you can create a procedurally defined scalar field (where you can type in some arithmetic expression) by choosing *Scalarfield* from the *Data* sub-menu. The icon of a newly created object usually will not be connected to any other object in the Pool. In order to establish connections later on, use the popup menu over the small white square on the left side of the object's icon. You can also put in links to scripts in the *Create* menu. Details are defined in Section 5.5 (Configuring popup menus).

3.1.5 View Menu

The *View* menu provides control over several *Viewer* options affecting the display independent of the *Viewer* input.

3.1.5.1 Layout

The *Layout* button lets you select between one, two, or four 3D viewers. All viewers will be placed inside a common window using a default layout. If you want to create an additional viewer in a separate window, choose *Extra Viewer*. You may create even more viewers using the Tcl command `viewer <n> show`. Starting from $n=4$, viewers will be placed in separate windows.

3.1.5.2 Background

The *Background* button opens the background dialog, allowing you to switch between the different background styles *uniform*, *gradient*, and *checkerboard*. In addition, the dialog allows you to adjust the two colors used by these styles.

In order to change the background color via the command interface, use the viewer commands `viewer <n> setBackgroundColor` and `viewer <n> setBackgroundColor2`. The command interface also allows you to place an arbitrary raster image into the viewer background (see Section 5.3.3.1, viewer commands).

3.1.5.3 Transparency

The *Transparency* button controls the way of calculating pixel values with respect to object transparencies during the rendering process.

- *Screen Door*: Transparent surfaces are approximated using a stipple pattern.
- *Add*: Additive alpha blending.
- *Add Delayed*: Additive alpha blending with two rendering passes. Opaque objects come first and transparent objects come second.

- *Add Sorted*: Like *Add Delayed*, but transparent objects are sorted by distances of bounding box centers from the camera and are rendered in back-to-front order.
- *Blend*: Multiplicative alpha blending.
- *Blend Delayed*: Multiplicative alpha blending with two rendering passes. Opaque objects come first and transparent objects come second.
- *Blend Sorted*: Like *Blend Delayed*, but transparent objects are sorted by distances of bounding box centers from the camera and are rendered in back-to-front order.
- *Sorted Layers*: Uses a fragment-level depth sorting technique, which gives better results for complex transparent objects. Multi-Texture, Texture Environment Combine, Depth texture, and Shadow OpenGL extensions must be supported by your graphics board. If the graphics board does not support these extensions, behaves as if *Blend Sorted* was set.

3.1.5.4 Lights

The *Lights* menu lets you activate different light settings for the 3D viewer. By default, the viewer uses a single headlight, i.e., a directional light pointing in almost the same direction as the camera is looking. The headlight can be switched on or off in each viewer via the viewer's popup menu. Alternatively, the headlight can be switched on or off for all viewers using the headlight toggle in this *Lights* menu. This standard light settings can be restored using the *Standard* button. More light settings can be defined by creating an appropriate file in `$AMIRA_ROOT/share/lights`. By default, *amira* provides one additional light setting including colored lights (*BlueRed*).

At any time, additional lights can be created via the *Create light* option. Except for the viewer's default headlight, all lights are represented by little blue icons in the Pool, just like ordinary data objects or modules. In order to make all hidden light icons visible, use the *Show all icons* option. *Hide all icons* hides the icons of all light objects. For more information about *lights*, please refer to the Reference Section of this manual.

3.1.5.5 Fog

The *Fog* button introduces a fog effect into the displayed scene and controls how opacity increases with distance from the camera. The fog effect will only be seen on a *uniform* background. More fine tuning is provided by the `fogRange Viewer command`.

- *None*: No fog effect (default).
- *Haze*: Linear increase in opacity with distance.
- *Fog*: Exponential increase in opacity with distance.
- *Smoke*: Exponential squared increase in opacity with distance.

3.1.5.6 Axis

The *Axis* button creates an *Axis* module named *GlobalAxis* which immediately displays a coordinate frame in the viewer window. This button is a toggle, so clicking on it again deletes the *GlobalAxis* module and removes the coordinate frame from the viewer window. The axes will be centered at the origin of the world coordinate system. You may also create local axes by selecting the appropriate entry from a data object's popup menu.

3.1.5.7 Measuring

The *Measuring* button creates an instance of a *Measuring* module that lets you measure distances and angles on objects within the viewer.

3.1.5.8 Easy Fade

The *Easy Fade* toggle lets you switch on a fading effect which is applied to all kinds of scene movements. Before a new image is rendered only a certain fraction of the background will be cleared. In this way older images remain visible until they fade out after a while. Note that this mode requires single buffer rendering, and therefore, flickering may be visible in some cases.

3.1.5.9 Frame Counter

The *Frame Counter* toggle lets you switch on a frames-per-second counter that will be displayed in the first viewer (viewer 0).

3.1.5.10 Console

The *Console* toggle lets you switch on or off display of the *Console window*.

3.1.6 Online Help

amira user's documentation is available online. You can access it via the *User's Guide* entry of the main window's *Help* menu. The user's guide contains some introductory chapters, as well as a reference section containing documentation for specific

- modules,
- data types,
- editors,
- file formats,
- and other components.

You may access the documentation of any such object via a separate index page accessible from the home page of the online help browser. *amira* modules also provide a question mark button in the Properties Area. Pressing this button directly pops up the help browser for the particular module.

By default, the help browser is displayed in the lower left portion of your *amira* window, in the same pane as the console. A tab button allows you to switch between the two. To request that the help browser to be displayed in its own top-level window, use the *Layout* tab of the *Edit/Preferences* menu.

Going through the online documents is similar to text handling within any other hypertext browser. In fact, the documentation is stored in HTML format and can be read with a standard web browser as well. Some specially marked (colored and underlined) text items allow you to jump quickly to related or referenced topics, where blue items point to unread sections, and red items to already viewed sections. Use the *Backward* and *Forward* buttons to scroll in the document history and *Home* to move to the first page.

Searching the online documentation

The online help browser provides a very simple interface for a full text search. First, press the *Question Mark* button to display the *Search* dialog. Then enter the desired text into the text field. Press the *Next* button in order to perform the search.

For example, suppose you are looking for information about the *surface editor*. Here are the results of various search strings you might enter:

- **Search string:** surface editor
Result: all occurrences of "surface" *or* "editor"
- **Search string:** surface +editor
Result: all occurrences (pages) containing the word "surface" *and* the word "editor"
- **Search string:** "surface editor"
Result: all occurrences of the string (case insensitive) of "surface editor"

Radio buttons allow you to search the *current page* or throughout *all documentation*. In *current page* mode, each time you press the *Next* button, the next occurrence of the search string is highlighted.

When the help browser is shown in a top-level window, its user interface is slightly different. You enter the search string into the text field in the upper part of the help browser window. Then you press either the *Search* button to search the entire document, or you press the *Find* button to search the current page. Use the *Backward* and *Forward* buttons to scroll in the document history and *Home* to move to the first page.

Running demo scripts

In the demo section of the on-line manual you can easily start any demonstration just by clicking on the marked text. The script will be loaded and executed immediately. You may interrupt running demo scripts by using the stop button in the lower right of the *amira* main window.

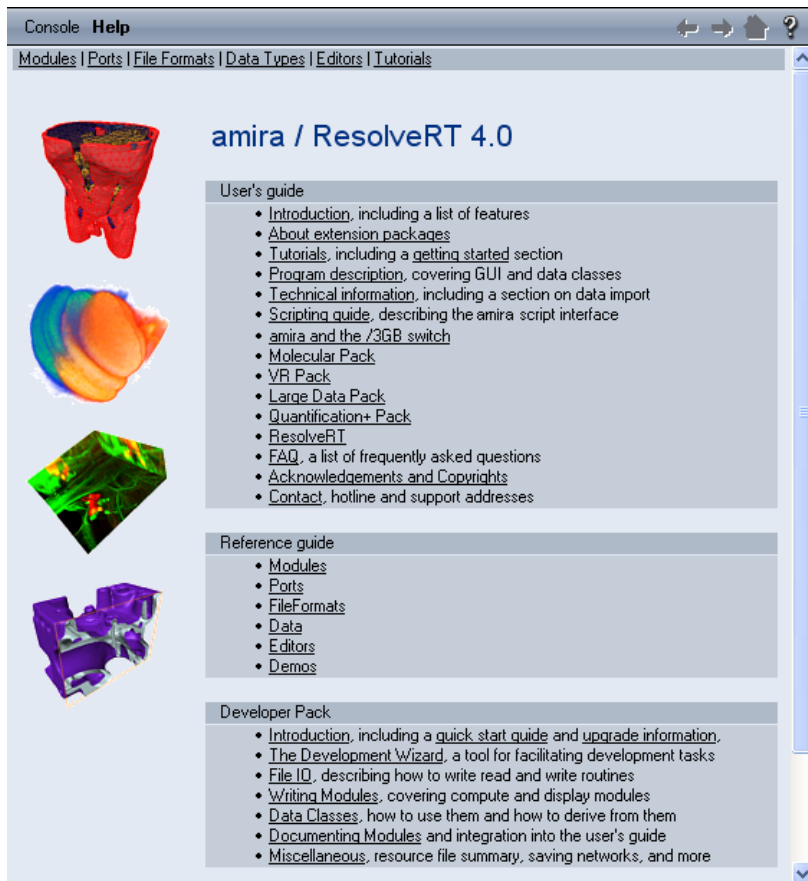


Figure 3.1: amira's help window.

Commands

`help`

Makes the help dialog appear and loads the home page of the online help.

`help getFontName`

Returns the name of the font of the browser.

`help setFontName`

Sets the font of the browser.

`help getFontSize`

Returns the size of the font of the browser.

`help setFontSize`

Sets the size of the font of the browser. In order to permanently change the font size, put this command in the `.amira` file in your home directory or in an `amira.init` file in the current working directory. For details see Section [4.4](#).

`help load file.html`

Load the specified hypertext document in the file browser. Note that only a subset of HTML is supported.

`help reload`

Reload the current document.

3.1.7 Main Window

amira's Main window consists of two components, the *Pool* in the upper part of the window and the *Properties Area* in the lower part. The *Pool* contains icons representing data objects and modules currently in use as well as lines connecting icons indicating dependencies between objects and modules. The *Properties Area* is the place where the user interface of selected objects is displayed. Typically, the interface consists of buttons and sliders arranged in *Ports*. They can be thought of as ports because the user can pass information to a module through them. The *Pool* and the *Properties Area* are described below.

By default, the Main Window, the *Viewer window*, the *Console window*, and the *Help browser* are all panes of a single large *amira* window. However, it is possible to designate that any of these last three components be displayed in its own top-level window using the *Layout* tab of the *Edit/Preferences* menu.

3.1.7.1 Pool

Concepts

Once a data object has been loaded or a module has been created, it will be represented by an icon in

the Pool. Some objects, especially initially loaded colormaps, may not be visible here. Such hidden objects are listed in the *Pool/Show Object* menu of the Main Window. Selecting an object from this menu causes the corresponding icon to be made visible in the Pool.

Icon colors are used indicate different types of objects. Data objects are shown in green and are the only objects which can be saved to disk using *File/Save*. Computational modules are shown in red, visualization modules are yellow, and visualization modules of *slicing* type are orange. Such modules may be used to clip the graphical output of any other module.

Connections between data objects and processing modules, shown as blue lines, represent the flow of data. For display modules, these connections show the data used to generate the display. You may connect or disconnect objects by picking and dragging a blue line between object icons.

As you might expect, not all types of processing modules are applicable to all kinds of data objects. If you click on a data object icon with the right mouse button, a menu pops up that shows all types of modules that can be connected to that object. (Alternatively, you can click on the small white triangle on the right side of the icon with the left, right, or middle button.) Selecting one of the modules will automatically create an instance of that module type and connect it to the data object. A new icon and a connecting line will appear in response. This way you can set up a more or less complex network that represents the computational steps required to carry out a specific visualization task and is used to trigger them. Note that modules used will appear as shortcut (or macro) buttons in the upper part of the window.

If you look closer at an object's icon you will notice two small squares on its left, one white and one orange. If you click on the white square with the left (or right, or middle) mouse button, a menu pops up that shows all connection ports of that object.

If the white square has a "-" or a "+" inside, you can click it with the left (or right, or middle) mouse button to hide ("collapse") or unhide the objects connected to that object. The collapse feature is helpful if there are too many objects in the Pool to view easily at once. The hidden objects will be visible in the *Pool/Show Object* menu.

The orange square controls the object's visibility in the viewers. It shows the current viewer layout (1 viewer, 2 viewers, etc.). Click inside the region of the square corresponding to a specific viewer to toggle the visibility of the object in that viewer. The region turns gray when the module is set to invisible. If you are using more than 4 viewers, each additional viewer will have its own orange square on the object's icon. You can also control an object's visibility by clicking on its visibility toggle in the Properties Area.

As mentioned above, for most objects the required connections are automatically established on creation. However, in order to set up optional connections you must use the connection popup menu. For example, you may attach an optional scalar field to an Isosurface module's Colorfield port in order to color the surface using the values in the Colorfield.

Once you have selected an entry from the connection popup menu of the object icon, you can choose a new input object for that port. In order to do so, click on the input object's icon in the Pool. The blue connection line will become lighter blue if the connection port can be connected to the chosen object.

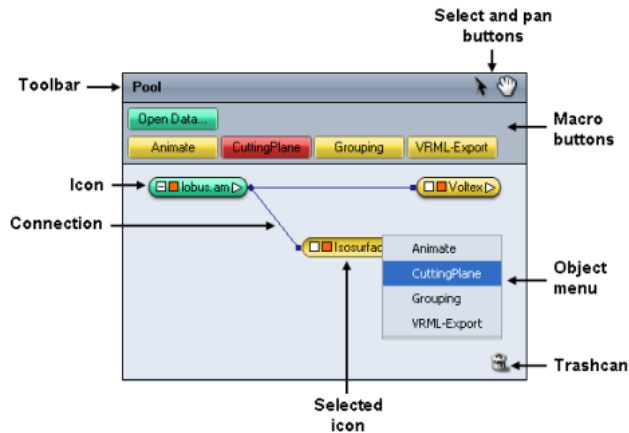


Figure 3.2: The Pool contains data objects and module icons.

Reasons for not being able to connect an input to a port can include the following: incompatible data type (use Castfield to change), incompatible dimensionality of the input (use ChannelWorks to change), incompatible grid type of the data (for example, use Arithmetic to sample on a regular grid), or simply that the connection does not make sense, for example, connecting a display module for surfaces to a CT data set.

In order to disconnect an input object click on the icon of the module the port belongs to. Data objects possess a special connection port called *Master*. This port refers to a computational module or editor the data object is attached to. It indicates that the computational module or editor controls the data object, i.e., that it may modify its contents.

Each object has an associated control panel containing buttons and sliders for setting or changing additional parameters of the object. The control panel becomes visible once the object has been selected, i.e. by clicking on its icon with the left mouse button. In order to select multiple objects, you can shift-click the corresponding icons or select them by using the rectangular selection tool (press and hold down the left mouse button over the Pool background, then sweep out a rectangle). Clicking on the icon of a selected object deselects it again. Clicking somewhere on the background of the Pool causes all selected objects to be deselected. One or more selected icons may be dragged around in the Pool by clicking on them and moving the mouse pointer while holding down the mouse button.

Interface

At the right of the Pool banner are the *PoolSelect* (arrow) and *PoolPan* (hand) buttons. In *PoolSelect* mode, the mouse is used for selecting, positioning, connecting, and disconnecting objects in the Pool. In *PoolPan* mode, moving the mouse pans the Pool workspace. Press the Control key to switch temporarily from one mode to the other.

At the top of the Pool is a region containing shortcut (or macro) buttons. The *Open Data* button is

a shortcut to the *File/Open Data* menu item. Up to 4 additional buttons are automatically displayed depending on the currently selected object(s). They provide easy access to the modules that are most commonly used and/or that have been recently used with the currently selected object.

The trashcan at the bottom right of the Pool provides a convenient shortcut for removing an object. Drag the object to the trashcan. When the cursor turns into an "X", release the mouse button and the object will be removed from the Pool. If you remove a data object, all modules downstream from that module will be deleted as well. Alternatively, you can use the *Remove object* item from the *Pool* menu, or you can use the Delete key or Ctrl-x to remove a selected object.

3.1.7.2 Properties Area

Once an object has been selected, its input controls will be displayed in the Properties Area below the Pool. Each object has a specific set of controllable parameters or options. These are described in detail for each module in the *index section* of the reference manual. Computational modules and visualization modules also provide a question mark button which lets you access the documentation of that module directly.

The *Apply* button is active (green) for modules that require you to explicitly initiate their action. Typically this is the case for modules whose action may take a significant amount of time, such as some of the compute modules. When the *Stop* button is red, you can press it to interrupt the action.

Check on the *auto-refresh* check box to automatically force an apply for any change in the network.

At the top of an object's control panel its name is displayed and a number of additional control buttons are provided. All objects have one or more orange viewer buttons for each 3D viewer. These buttons control whether any graphical output of an object is displayed in a particular viewer or not. For example, if you have two viewers and two isosurface modules you may want to display one isosurface in each viewer.

Display modules of *slicing* type (orange ones) provide a clip button. Clicking this button will cause the graphical output of any other module to be clipped by that slice. Clipping does not affect modules with hidden geometry or modules that are created after the clip button has been pressed.

Data objects provide a number of additional *editor* buttons. Editors are used in order to modify the contents of a data object interactively. For example, you can perform manual segmentation of 3D image data by editing *label fields* using the *image segmentation editor*. Some editors display their controls in the Properties Area like all other objects, while others use a separate dialog window that allows you to perform object manipulations.

As already mentioned, specific input controls of an object or a module are organized in *Ports*. Each port has a pin button on its left. If a port is pinned it will still be visible even when the object is deselected. The ports are composed of various widgets that reflect an operational meaning, e.g., a value is entered by a slider, a state is set by radio buttons, a binary choice is presented as a toggle button. The control elements have a uniform layout and are divided into several basic types. A description of the basic port types is contained in the *component index* section of the User's Reference Manual.

3.1.8 Viewer Window

The 3D viewer plays a central role in *amira*. Here all geometric objects are shown in 3D space. The 3D viewer offers powerful and fast interaction techniques. It can be regarded as a virtual camera which can be moved to an arbitrary position within the 3D scene. The left mouse button is used to change the view direction by means of a virtual trackball. The middle mouse button is used for panning, while the left and the middle mouse button pressed together allow you to zoom objects.

The virtual trackball controlled by the left mouse button allows for free rotation of the camera. The camera trackball, displayed by default in the lower right corner of the viewer, is used for constrained rotation: rotation of the camera about the screen-aligned X, Y, or Z axes. Click on the vertical wheel (it becomes red when you select it) and move the mouse up/down to rotate about the X axis. Click on the horizontal wheel (it becomes green when you select it) and move the mouse left/right to rotate about the Y axis. Click on the third wheel (it becomes blue) and move the mouse up/down to rotate about the Z axis.

Sometimes you need to manipulate objects directly in the 3D viewer. For example, this technique, called 3D interaction, is used by the *transform editor*. The editor provides special draggers that can be picked and translated or rotated in order to specify the transformation of a data object. Before you can interact with these draggers, you must switch the viewer into *interaction mode*. This is done by clicking on the arrow button in the upper left corner. If the viewer is in interaction mode, the mouse cursor will be an arrow instead of a hand symbol. You can use the [ESC] key in order to quickly switch between interaction mode and viewing mode. If the viewer is in interaction mode, use the [Alt] key to temporarily switch to viewing mode.

More than one viewer can be active at a time. Standard screen layouts with one, two, or four viewers can be selected via the *View menu*. Additional viewers can be created using the Tcl command `viewer <n> show`, where <n> is an integer number between 0 and 15. While viewers 0 to 3 will be placed in a common panel window, viewers 4 to 15 will create their own top-level window. For more specific control, the viewer provides an extensive command set, which is documented in Section 5.3.3.1.

The toolbar of the main viewer window provides several buttons and controls, see Figure 3.3. The precise meaning of these controls is described below.

Interact:



Switches the viewer into interaction mode. You can also use the [ESC] key to toggle between viewing mode and interaction mode.

Trackball:



Switches the viewer into viewing mode. You can also use the [ESC] key to toggle between

interaction mode and viewing mode. The left mouse button is used to change the view direction by means of a virtual trackball.

Translate:



Same as *Trackball* except that the left mouse button is used for panning (translation).

Zoom:



Same as *Trackball* except that in this mode vertical motion of the left mouse button controls zooming.

Rotate:



Rotates the camera around the current view direction. By default, a clockwise rotation of one degree is performed. If the Shift-key is pressed while clicking, a 90 degree rotation is done. If the Ctrl-key is pressed, the rotation will be counterclockwise.

Seek:



Pressing the seek button and then clicking on an arbitrary object in the scene causes the object to be moved into the center of the viewer window. Moreover, the camera will be oriented parallel to the normal direction at the selected point. Seeking mode may also be activated by pressing the [S] key in the viewer window.

Home:



Resets camera to the home position.

Set Home:



Sets the current position as the new home position.

Perspective/Ortho:



Toggles between a perspective and an orthographic camera. By default, a perspective camera is used. You may want to use an orthographic camera in order to measure distances or to exactly align objects

in 3D space. **Note:** Only one of these buttons will be visible at a time, the button indicating the currently active camera type.

View All:



Repositions the camera so that all objects become visible. The orientation of the camera will not be changed. **Note:** The first button is seen if *technical* naming has been selected in the *Edit/Preferences/Layout dialog*, the second button will be displayed if *medical* naming has been selected.

YZ-, XY- and XZ-Views:



Adjusts the camera according to the specified viewing direction. The viewing direction is parallel to the coordinate axis perpendicular to the specified coordinate plane. Medical doctors are used to viewing series of tomographic images parallel to the XY-plane with the y-axis pointing downwards. This convention is followed by the XY-button. The opposite view direction is used if the Shift key is pressed. **Note:** The first set of buttons is seen if *technical* naming has been selected in the *Edit/Preferences/Layout dialog*, the second set of buttons will be displayed if *medical* naming has been selected. They correspond to axial, coronal, and sagittal views respectively.

Stereo:



Allows you to enable or disable stereo viewing, as well as specify various stereo viewing parameters via the *Stereo Preferences* dialog.

Measuring:



Pressing this button creates an instance of a *Measuring* module that lets you measure distances and angles on objects within the viewer. Clicking on the down arrow will display a menu of measuring tools to choose from: *2D line*, *3D line*, *2D angle*, *3D angle*, and *annotation*. See the Measuring module's documentation for details. **Note:** Only one of these buttons will be visible on the toolbar at a time, the button of the measuring tool most recently accessed from the viewer toolbar.

Snapshot:



Takes a snapshot of the current rendering area and saves it in a file. The filename as well as the

desired output format must be entered through the *Snapshot dialog*. Snapshots may also be taken using the *viewer command snapshot*.

Layout:



Selects the viewer layout: a single view, two viewers side-by-side, two viewers stacked, or four viewers.

Fullscreen:



Selects fullscreen mode. In this mode, the viewer occupies the entire screen and no other windows will be visible. To exit fullscreen mode, click the right mouse button and uncheck *Fullscreen* in the popup menu.

In addition to these buttons, the *amira* viewers provide an extensive set of *Tcl commands*, which are listed in Section [5.3.3.1](#).

3.1.9 Console Window

The *console window* is a command shell allowing to access *amira*'s advanced control features. It serves two purposes. First, it gives you some feedback on what is currently going on. Such feedback messages include warnings, error indications and notes on problems as well as information on results. Second, it provides a command line interface where *amira* commands can be entered.

amira's console commands are based on the *Tcl* script language (*Tool Command Language*). Examples are:

```
load C:/MyData/something.am
viewer 0 setSize 200 200
viewer 0 snapshot C:/snapshot.tif
```

The *amira* scripting syntax and the specific commands are described in the Chapter [5](#) (Scripting). To execute a single console command just type in its name and arguments and press 'Enter'. If you select an object and then press the [TAB] key on the empty command line, then the name of the object will be automatically inserted.

You can also type the beginning of a command word and type the [TAB] key to complete the word. This only works if the beginning is unique. Pressing [TAB] a second time will show the possible completions. Often, this saves a lot of typing. Commands provided by data objects and modules are documented in the reference section of the users guide. Pressing the [F1] key for such a command without any arguments pops up the help text for this command. This is also true for commands provided by the *ports* of an object.

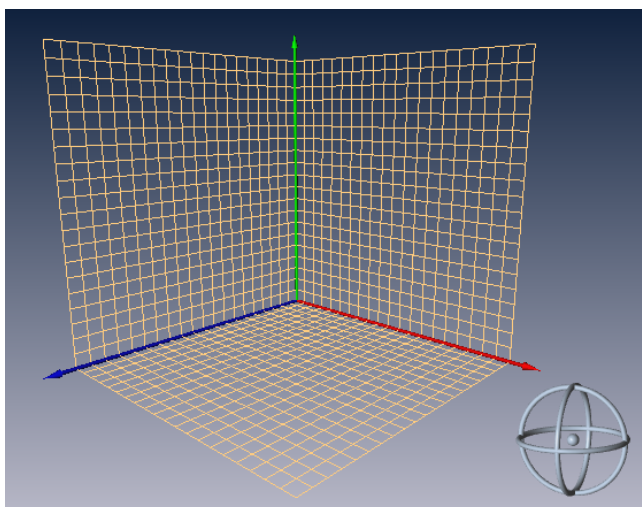


Figure 3.3: amira's viewer window provides a virtual trackball for easy navigation, as well as a camera trackball (lower right) for constrained rotation. The toolbar contains several controls, allowing you for example to switch between viewing mode and interaction mode, to choose certain orientations, or to take snapshots.

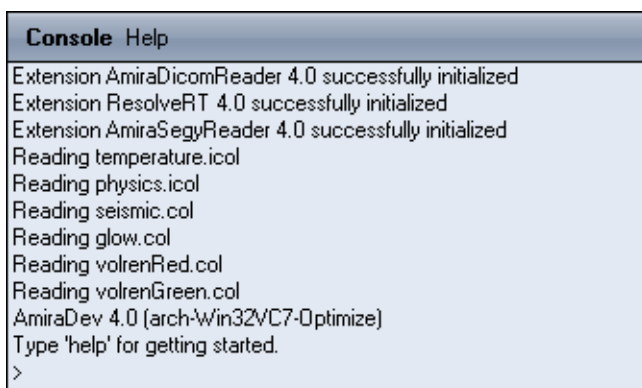


Figure 3.4: amira's console window displays info messages and lets you enter Tcl commands.

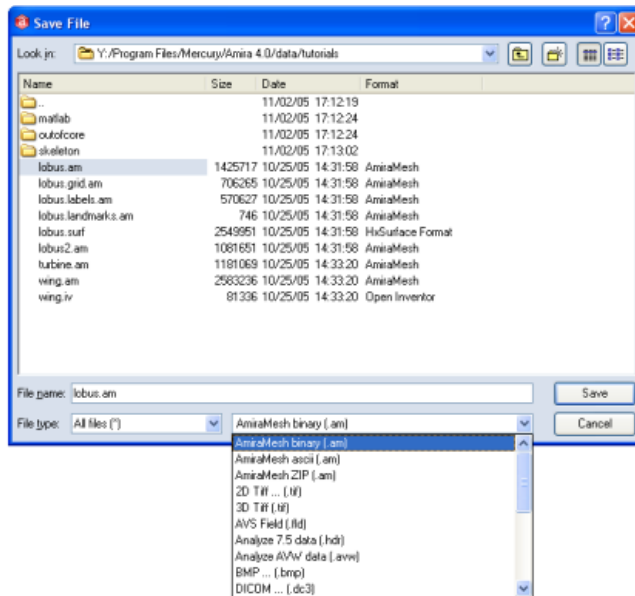


Figure 3.5: amira's file dialog.

Additionally the *console window* provides a command history mechanism. Use 'up arrow' and 'down arrow' to scroll up and down in the history list.

To execute a file containing many Tcl commands use `source <filename>` or load the script file via amira's file dialog from the file menu. amira script files are usually identified by the extension `.hx`. For advanced script examples take a look at amira's demo files located in `$AMIRA_ROOT/share/demo`.

3.1.10 File Dialog

The *File Dialog* is the user interface component for importing and exporting data into resp. from amira. It is used at several places in amira, most prominently by the *Load*, *Save Data As*, and *Save Network* items of the main window's *File* menu.

The dialog provides two modes of information, a detail mode and a multi-column mode. In the detail mode, which is active by default, some file data are shown next to each filename, namely the file size, the file's last modification time, and the file format. You may sort the file list according to each of these properties by clicking on the particular column's header bar. Subdirectories will always be displayed first. In multi-column mode only the file name is displayed. You may switch between both modes using the tool buttons in the upper right part of the dialog window.

Most file formats supported by **amira** will be recognized automatically, either by analyzing the file header or by looking at the file name suffix. A list of all *supported file formats* is contained in the reference section of this manual. You may manually set the format of a file by means of the dialog's popup menu (see below).

3.1.10.1 Changing Directories

You can change the current directory by double-clicking a subdirectory in the file list or by entering a new directory in the dialog's path list. By default, the path list contains the current directory, the directory containing the demo data sets provided with **amira**, as well as all directories defined by the environment variable `AMIRA_DATADIR`. In `AMIRA_DATADIR` multiple directory names have to be separated by colons [:] on Unix systems or by semicolons [;] on Windows systems. In addition, on Windows system the names of the twelve most recently visited directories are stored in the path list.

3.1.10.2 Selecting Files

To select a single file just click on it or type in its name in the file name text field.

In some cases you might want to select more than one file at once, e.g., when loading a 3D image data set as a series of single 2D images. You can do this by selecting the first file first and then shift-selecting the last file. Then all intermediate files will be selected as well. Moreover, you may ctrl-click a file in order to toggle its selection state individually.

3.1.10.3 Using the Filename Filter

The filename filter is visible when the dialog is in import mode (*Load File*). It is useful to restrict the list of filenames to a subset matched by the filter expression. The filter expression may contain the wildcard characters ? (matches any character) and * (matches an arbitrary character sequence). For example, the expression `*.img` matches all filenames with the suffix `.img`.

3.1.10.4 The File Dialog's Popup Menu

The file dialog provides a popup menu which may be activated by pressing the right mouse button over the file list. Among others, this menu lets you rename or delete files or directories, provided you have the permission to do that. Note that you may only delete empty directories.

Using the *Format* option of the popup menu you may manually set the format to be used when loading a file into **amira**. This option is useful if for some reason the wrong format has been detected automatically, or if no format at all could be detected. Note however, that any format specification set manually will be overwritten when the directory is reread the next time.

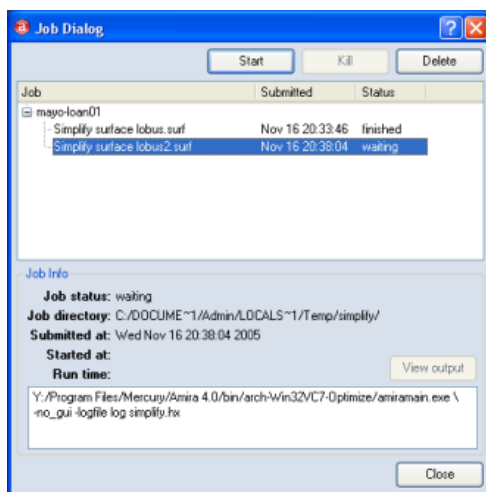


Figure 3.6: The job dialog lets you start, stop, examine, and delete batch jobs.

3.1.11 Job Dialog

Certain time-consuming operations in **amira** can be performed in batch mode. For this purpose **amira** provides a job queue, where jobs like generation of a tetrahedral grid can be submitted. You can inspect the current status of the job queue, start and delete jobs from the queue by selecting *Jobs* from **amira**'s file menu. This will bring up the *Job Dialog*.

In the upper part of the job dialog the current list of jobs of a user is shown. For each job a short description is displayed, as well as the time when the job has been submitted and the current state of the job. A job may be waiting for execution, running, finished, or it may have been killed.

The job directory

For each job a temporary directory is created containing any required input data, scripts, state information, and log files. On Unix systems this directory is created at the location specified by the environment variable `TMPDIR`. If no such variable exists, `/tmp` is used. On Windows systems the default temporary directory is used. Typically this will be `C:/TEMP`.

Controlling the job queue

A job's state may be manipulated using the action buttons shown above the job list. In order to start the job queue select the first job waiting for execution and then press the *Start* button. Note that only one job can be executed at a time. In order to kill a running job, select it in the job list and press the

Kill button. You may delete a job from the job queue using the *Delete* button. When deleting a job the temporary job directory will be removed as well.

Information about a job

Once you have selected a job in the job queue, more detailed information about it will be displayed in the lower part of the dialog window, notably the state of the job, the temporary job directory, the submit time, the time when the job has been started, the run time, and the name of the command to be executed. Any console output of a running job will be redirected to a log file located in the temporary job directory. Once such a log file exists and has non-zero size you may inspect it by pushing the *View output* button.

Commands

```
job submit cmd info [tmpdir]
```

Submits a new job to the job queue. *command* specifies the command to be executed. *info* specifies the info string displayed in the job dialog. *tmpdir* specifies the temporary job directory. If this argument is omitted a temporary job directory is created by *amira* itself. In any case, the directory will be automatically deleted when the job is removed from the job queue. Example: `job submit "clock.exe" "Test job"`

```
job run
```

Starts the first job in job queue pending for execution. When a job is finished, execution of the next job in the queue starts automatically, thus all jobs in the queue will be executed consecutively by `job run`.

3.1.12 Preferences Dialog

The *Preferences Dialog* allows you to adjust certain global settings of *amira*. The preferences are stored in a permanent fashion on a per-user basis. The dialog contains a tab bar with several tabs. The first tab, *General*, is related to the Pool and saving the network. The second tab, *Layout*, affects the layout of the user interface. The third, *On Exit*, controls what conditions are checked for in the networks when *amira* exits. The *Molecules* tab allows you to specify options for handling molecular data. The *LDA* tab allows you to specify options for out-of-core data. For advanced users, additional options can be set using the *LDAExpertSettings* module. The *Segmentation* tab allows you to specify options for the Segmentation Editor. See the *Segmentation Editor* documentation for details.

3.1.12.1 The General Tab

2-pass firing algorithm

If set, a slightly more complex firing algorithm is used which ensures that down-stream modules connected to an up-stream object via multiple paths are only fired once if the up-stream object changes. The default is off.

Auto-select new modules

If set, a new module selected from the popup menu of its parent object is shown automatically in the Properties Area. The default is on.

Deselect previously selected modules

This option can only be set if auto-selection is turned on. If set, all objects are deselected before selecting the new module. Otherwise the new module will be appended at the end of the Properties Area. The default is on.

Draw viewer toggles on icons

If set, small viewer mask toggles are drawn on the icons of data objects and display modules. This allows you to show or hide a module in a viewer without selecting it first. The default is on.

Draw compute indicator

If set, a small red rectangle is drawn inside the icon of a module to indicate that the module is currently working. The default is on.

Include unused data objects

If set, all data objects including hidden colormaps are stored in network scripts. When executing such a script all existing objects are removed first. If not set only visible data objects and objects which are referenced by others are stored in a network script. When executing the script hidden data objects are not removed. The default is on.

Include window sizes and positions

If set, window sizes and positions are stored in network scripts. Be careful using this option if you want to send your script to a machine with a different screen resolution.

Overwrite existing files in auto-save

If set, no overwrite check is performed for data objects which need to be saved automatically in order to create a network script. Otherwise a unique file name will be chosen. The default is on. Details about the auto-save feature are described in Section [3.1.1.7](#).

3.1.12.2 The Layout Tab

Naming convention

These toggles allow you chose between two naming conventions: *medical* and *technical*. Your choice will affect the labels of some ports, for example, Orientation ports, as well as the appearance of the viewer buttons controlling the view orientation.

Camera trackball

The camera trackball, used for constrained rotation of the camera about the screen-aligned X, Y, or Z axes, is described in Section [3.1.8](#).

Show the trackball

This toggle controls the visibility of the camera trackball. The default is on.

Auto-hide the trackball

When this box is checked, the trackball is only displayed while the mouse is within the trackball display area. It is hidden as soon as the mouse moves outside the trackball display area. The default is off. This item is not active if the *Show the trackball* item is off.

Position

This menu allows you to specify which corner of the viewer window to display the trackball in. The default is *Lower right*. This item is not active if the *Show the trackball* item is off.

Windows

These items allow you to configure the layout of the **amira** windows. By default, the main **amira** interface components (Pool, Properties Area, main Viewer window, Console, and Help Browser) share one large window. You can use the check boxes to display some of these components in separate top-level windows. This gives you additional flexibility in managing the "real-estate" of your graphics display. For example, on a dual-head display it can be interesting to display the main Viewer window on one display and the rest of the **amira** interface on the other.

Show viewer window in top-level window

The main Viewer window will be displayed in a top-level window.

Show console in top-level window

The console will be displayed in a top-level window.

Show help browser in top-level window

The help browser will be displayed in a top-level window.

Show "DoIt" buttons

Some modules have a button, usually labeled "DoIt", to initiate the action of the module. Since **amira** 4.0, by default, this button is not displayed in the Properties Area. Rather, the green *Apply* button at the bottom of the Properties Area is used to initiate the action. If this box is checked, the button will be displayed in the Properties Area. The green *Apply* button will still be available for use as well.

Finally, you have the choice to *save window layout on exit*, *Save current layout (now)*, and *Restore current layout (now)*.

3.1.12.3 The On Exit Tab

The page allows you to control what conditions are checked for in the networks when **amira** exits.

3.1.12.4 The Molecules Tab

Color Schemes

These check boxes allow you to chose alternate color schemes: CPK for atoms, and RasMol for amino acids.

Selection Info

These items control how much information is printed into the console when parts of a molecule are

selected. Activate *Molecule name* if the name of the molecule should be printed. Activate *Group name* if the name of the selected group should be printed. If you activate *Group attributes*, all attributes of the selected group are printed. If *Explicit attributes* is activated, the printed attributes are restricted to those explicitly named in the corresponding text field.

Atom Expressions

3.1.12.5 The LDA Tab

Out-of-core threshold

Specifies the size above which data sets will be treated as out-of-core data.

Main memory amount

Sets the maximum allowed main memory in MB (megabytes) for all the out-of-core data sets.

Video memory amount

Sets the maximum allowed texture memory in MB (megabytes) for all the out-of-core data sets.

Viewpoint refinement

If set, refinement depends on the viewpoint.

View culling

If set, refinement takes place only in the view frustum.

Move low resolution

If set, ortho slices and oblique slices are computed in half resolution when moving.

Loading policy

Sets loading behavior. If *No Interaction* is selected, the asynchronous loading thread will only load when the user does not interact with the scene. If *Always* is selected, loading occurs as long as there is something to load. If *Never* is selected, no loading occurs. The default is *No Interaction*.

3.1.12.6 The Segmentation Tab

Material

If set, materials are shown in the 3D viewer. This toggle applies when material draw style *3D view* is enabled in the Segmentation Editor. Otherwise, it has no effect.

Slice position

If set, the slice position is displayed in the 3D viewer.

Show 3D

If set, the selection is shown in the 3D viewer.

Draw style

This option lets you choose the material draw style used in the 3D viewer.

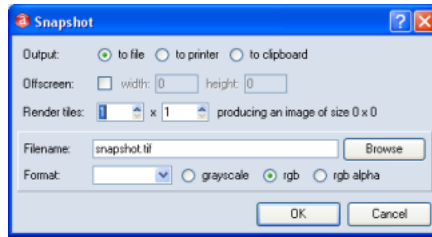


Figure 3.7: The snapshot dialog allows you to save or print the contents of a viewer window.

3.1.13 Snapshot Dialog

The *Snapshot Dialog* provides the user interface of the viewer’s snapshot facility. You get the dialog by clicking on the camera icon in the *viewer* toolbar.

- **Output:** Specifies the output device. With *to file* the grabbed image is saved to a file, with *to printer* the image is sent directly to the printer, and with *to clipboard* it is sent to the clipboard. In the *to printer* mode you first must select and configure a printer by pressing the *Configure* button. In addition, you may enter an arbitrary text string which is printed as an annotation text below the snapshot image.
- **Offscreen:** Lets you grab images larger than the actual screen size. When this option is checked, the output dimensions can be specified in the *width* and *height* text fields up to a maximum of 2048 x 2048 pixels.
- **Render tiles:** Use this option to render snapshots of virtually unlimited resolution (e.g. for high quality printouts). In this mode the scene is divided into $n \times m$ tiles where n and m can be entered into the adjacent text fields. Then the camera position is set such that each tile fills the current viewer and a snapshot is taken. Finally the tiles are internally merged to a single image and sent to the device specified in the *Output* port.
- **Filename:** Lets you specify the filename if the *to file* option is set. The *Browse* button allows you to browse to a desired location within the filesystem.
- **Format:** The format option lets you select the file format to be produced for file output. The following formats are supported: TIFF (.tif, .tiff), SGI-RGB (.rgb, .sgi, .bw), JPEG (.jpg, .jpeg), PPM (.pgm, .ppm), BMP (.bmp), PNG (.png), DCM (.dcm), and Encapsulated PostScript (.eps). In addition, this port offers three radio buttons to choose between *grayscale*, *rgb*, and *rgb alpha* type of raster images. If *rgb alpha* option is set, images are produced such that the viewer background is assigned to the alpha channel. This option is not available for file formats that do not support an alpha channel.

3.1.14 System Information Dialog

The system information dialog provides diagnostics information allowing the user or the **amira** support team to better analyze software problems. The dialog contains a tab bar with three pages. The first page lists information about the current CPU. The second page lists information about the current OpenGL graphics driver. The third page lists version information about the currently installed **amira** components. In the lower left part of the dialog you will find a button *Save Report*. With this button all information can be written into a text file. In case of a support call, you may be asked to send this text file to the hotline.

3.1.14.1 The CPU Tab

This page displays information about the system on which you are running **amira**. This information can be useful when reporting problems to technical support.

3.1.14.2 The OpenGL Tab

This page displays information about the current OpenGL graphics driver. In particular, a list of available OpenGL extensions is printed. This list allows it to check if certain rendering techniques like direct volume rendering via 3D textures are supported on a particular hardware platform or not.

3.1.14.3 The Libraries Tab

This page displays a list of all DLLs or shared libraries contained in the **amira lib** directory. For each library certain version information as well as an MD5 checksum are printed. In this way it is possible to check whether a certain patch has already been installed or not. For most libraries the version information is compiled in. For other libraries this information is read from the version information files found in `share/versions` in the **amira** root directory.

3.2 General Concepts

This section contains some general comments on how data objects are organized and classified in **amira**. In particular, the following topics are discussed:

- *amira Class Structure*
- *Scalar and Vector Fields*
- *Coordinates and Grids*
- *Surface Data*
- *Vertex Set*
- *Transformations*
- *Parameters*

3.2.1 Class Structure

In this section we discuss the object-oriented design of *amira* in a little more detail. You already know that data objects, e.g., gray level image data or vector field sets, appear as separate icons in the *Pool*. You also know that there are certain display modules which can be used to visualize the data objects. While some modules can be connected to many different data objects, e.g., the *Bounding Box* module, others cannot, e.g., the *OrthoSlice* module. The latter can only be connected to voxel data or to scalar distributions on voxel grids. The reason is that internally both are represented as a scalar field with uniform Cartesian coordinates. Consequently, the same visualization methods can be applied to both. On the other hand, for example a volumetric tetrahedral grid model of the object of interest usually looks completely different. But since it is also a 3D data object, the same *Bounding Box* module can be connected to it.

In summary, there are *amira* data objects that might be conceived of different type, but with respect to mathematical structure, applicability of viewing and other processing modules, as well as programming interface design have many common properties. Obeying principles of object-oriented design, the data types of *amira* are organized in class hierarchies where common properties are attributed to 'higher up' classes and inherited to 'derived' classes, as sub-classes of a class are commonly referred to. Conceptually each object occurring in *amira* is an instance of a class and each of its predecessors in the hierarchy that the class belongs to. The classes and their hierarchies are defined within *amira*. As the user you normally deal with instances of classes only. For instance, there is a class called "HxObject" with sub-classes "HxData" and "HxModule". "HxData" comprises the types of data associated with data objects used for modeling the objects of interest, e.g., volumetric tetrahedral grids or surfaces. "HxModule" comprises data types that have been assigned to display and other processing modules, again in accordance with principles of object-oriented design. This is why *amira*'s data objects and processing modules are commonly referred to as "objects".

There are also classes in *amira* that are not derived from "HxObject" and constitute other data types, and there are several independent class hierarchies. e.g., there is a class called "HxPort" from which all classes supporting the operation and display of interface control elements are derived (see section *Properties Area* and the *List of Ports* in the index section of the user's guide).

A single class hierarchy is usually figured as an upside-down tree, i.e. with the root at the top. Thus the *data* class tree is the one to which the information as to which processing module is applicable to which data object is hooked. Its classes reflect the mathematical structure of the object models supported by *amira*. For example, scalar fields and vector fields are such structures and derived from a common "field" class which represents a mapping $R^3 \rightarrow R^n$. Deriving a sub-class from this base class requires a value to be specified for n .

At the same time fields defined on Cartesian grids are distinguished from fields defined on tetrahedral grids, i.e., this distinction is part of the classification scheme that gives rise to branches in the *data* class subtree. In the next section of this chapter you will learn more about the *data* class hierarchy. In the second section we discuss how some data types frequently used for various visualization tasks fit into it.

Internally, all class names begin with a prefix *Hx*. However, you don't have to remember these names

unless you want to use the command shell to create objects. For example, a bounding box is usually created by choosing the *BoundingBox* item from the pop-up menu of a data object that is to be visualized, but you may also create it by typing `create HxBoundingBox` in the command window.

3.2.2 Scalar Field and Vector Fields

The most important fields in *amira* are three-dimensional ones. These fields are defined on a certain domain $\subseteq \mathbb{R}^3$. A field can be evaluated at any point inside its domain. If the field is defined on a discrete grid, this usually involves some kind of interpolation.

3.2.2.1 Scalar Fields

A 3D scalar field is a mapping $\mathbb{R}^3 \rightarrow \mathbb{R}$. The base class of all 3D scalar fields in *amira* is *HxScalarField3*. Various sub-classes represent different ways of defining a scalar field. There are a number of visualization methods for them, for example pseudo-coloring on cutting planes, iso-surfacing, or volume rendering. However, many visualization modules in *amira* rely on a special field representation. Therefore, they can only operate on sub-classes of a general scalar field. Whenever a given geometry is to be pseudo-colored, any kind of scalar field can be used (cf. *Colorwash*, *GridVolume*, *Isosurface*).

The class *HxTetraScalarField3* represents a field which is defined on a tetrahedral grid. On each grid vertex a scalar value, e.g., a temperature, is defined. Values associated to points inside a tetrahedron are obtained from the four vertex values by linear interpolation. This class does not provide a copy of the grid itself, instead a reference to the grid is provided. This is indicated in the Pool by a line which connects the grid icon and the field icon. As a consequence, a field defined on a tetrahedral grid cannot be loaded into the system if the grid itself is not already present.

The class *HxRegScalarField3* represents a field which is defined on a regular Cartesian grid. Such a grid is organized as a three-dimensional array of nodes. In the most simple case these nodes are axis-aligned and have equal spacings. The coordinates of such a uniform grid can be obtained from a simple bounding box containing the origin vector and increments for all directions. Stacked coordinates are another example. Here the spacing in z-direction between subsequent slices may be different. In any case scalar values inside a hexahedral grid cell are obtained from the eight vertex values using trilinear interpolation. While the *OrthoSlice* module can only be used to visualize scalar fields with uniform or stacked coordinates, other modules like *ObliqueSlice* or *Isosurface* work for all scalar fields with regular coordinates.

Yet another example of a scalar field is the class *HxAnnaScalarField3*. It represents an analytically defined scalar field. To create such a field, select *ScalarField* from the *Create* menu of *amira*'s main window. You have to specify a mathematical expression which is used to evaluate the field at each requested position. Up to three other fields can be connected to the object. These can be combined to a new scalar field, even if they are defined on different grids.

3.2.2.2 Vector Fields

As for scalar fields `amira` provides a number of vector field classes, these are derived from the base classes `HxVectorField3` and `HxComplexVectorField3`. While ordinary vector fields return a three-component vector at each position, complex vector fields return a six-component vector. Complex vector fields are used for encoding stationary electromagnetic wave pattern as required by some applications. Usually complex vector fields are visualized by projecting them into the space of reals using different phase offsets. The `Vectors` module even allows you to animate the phase offset. In this way a nice impression of the oscillating wave pattern is obtained.

3.2.3 Coordinates and Grids

`amira` currently supports two important grid types, namely grids with hexahedral structure (regular grids), and unstructured tetrahedral grids. Other types, e.g., unstructured grids with hexahedral cells or block-structured grids will be added in future releases of `amira`.

3.2.3.1 Regular Grids

A regular grid consists of a three-dimensional array of nodes. Each node may be addressed by an index triple (i,j,k) . Regular grids are further distinguished according to the kind of coordinates being used. The most simple case comprises *uniform* coordinates, where all cells are assumed to be rectangular and axis-aligned. Moreover, the grid spacing is constant along each axis. A grid with *stacked* coordinates may be imagined as a stack of uniform 2D slices. However, the distance between neighboring slices in z-direction may vary. In case of *rectilinear* coordinates the cells are still aligned to the axes, but the grid spacing may vary from cell to cell. Finally, in case of *curvilinear* coordinates each node of the grid may have arbitrary coordinates. Grids with curvilinear coordinates are often used in fluid dynamics because they have a simple structure but still allow for accurate modeling of complex shapes like rotor blades or airfoils.

3.2.3.2 Tetrahedral Grids

The `TetraGrid` class represents a volumetric grid composed of many tetrahedrons. Such grids can generally be used to perform finite-element simulations, e.g., E-field simulations.

A considerable amount of information is maintained in a `TetraGrid`. For each vertex a 3D coordinate vector is stored. For each tetrahedron the indices of its four vertices are stored as well as a number indicating the segment the tetrahedron belongs to as obtained by a segmentation procedure. Beside this fundamental information a number of additional variables are stored in order for the grid being displayed quickly. In particular all triangles or faces are stored separately together with six face indices for each tetrahedron. In addition for each face pointers to the two tetrahedrons it belongs to are stored. This way the neighborhood information can be obtained efficiently.

When simulating E-fields using the finite-element method, the edges of a grid need to be stored explicitly, because vector or Whitney elements are used. These elements and its corresponding coefficients

are defined on a per-edge basis. When a grid is selected information on the number of its vertices, edges, faces, and tetrahedrons is displayed.

3.2.4 Surface Data

amira provides a special-purpose data class for representing triangular surfaces, called *HxSurface*. This class is documented in more detail in the index section of the user's guide. For the moment, we only mention that the class maintains connectivity information and that it may represent manifold as well as non-manifold topologies.

The surface class provides a rich set of Tcl commands. It is a good example of an **amira** data class that does not simply store information, but allows the user to query and manipulate the data by means of special-purpose methods and interfaces.

3.2.5 Vertex Set

Another example of data abstraction and inheritance is the *VertexSet* class. Many data objects in **amira** are derived from this class, e.g., landmark sets, molecules, surfaces, or tetrahedral grids. All these objects provide a list of points with x-, y-, and z-coordinates. Other modules which require a list of points as input only need to access the *VertexSet* base class, but don't need to know the actual type of the data object.

One such example of a generic module operating on *VertexSet* objects is the *VertexView* module. This module allows you to visualize vertex positions by drawing dots or little spheres at each point.

3.2.6 Transformations

Data objects in **amira** can be modified using an arbitrary affine transformation. For example, this makes it possible to align two different data objects so that they roughly match each other. Internally, affine transformations are represented by a 4x4 transformation matrix. In particular, a uniform scalar field remains a uniform scalar field, even if it is rotated or sheared. Display modules like *OrthoSlice* still can exploit the simple structure of the uniform field. The possible transformation is automatically applied to any geometry shown in the 3D viewer.

In order to interactively manipulate the transformation matrix use the *Transform Editor* (documentation is contained in the index section of the user's guide).

Be careful when saving transformed data sets! Most file formats do not allow to store affine transformations. In this case you have to apply the current transformation to the data. This can be done using the Tcl-command `applyTransform`. In case of *vertex set objects* the transformation is applied to all vertices. Old coordinates are replaced by new ones, and the transformation matrix is reset to identity afterwards. After a transformation has been applied to a data set, it cannot be unset easily anymore.

If a transformation is applied to uniform fields, e.g., to 3D image data, the coordinate structure is not changed, i.e., the field remains a uniform one. Instead, the data values are resampled, i.e., the

transformed field is evaluated at every vertex of the final regular grid. The bounding box of the resulting grid is modified so that it completely encloses the transformed original box.

3.2.7 Parameters

For any data object an arbitrary number of additional parameters or attributes may be defined. Parameters can be interactively added, deleted, or edited using the *parameter editor*. Parameters are useful for example to store certain parameters of a simulation or of an experiment. In this way the history of a data object can be followed.

There are certain parameters which are interpreted by several **amira** modules. The meaning of these parameters is summarized in the following list:

- `Colormap name`
This specifies the name of the default colormap used to visualize the data. Some modules automatically search the Pool for this colormap and for example use it for pseudocoloring.
- `DataWindow minVal maxVal`
This indicates the preferred data range used for visualizing the data. The `OrthoSlice` module automatically maps values below `minVal` to black and values above `maxVal` to white.
- `LoadCmd cmd`
This parameter is usually set by import filters when a data object is read. It is used when saving the current network into a file and it allows to restore the object automatically. Internal use only.

Note that there are many file formats which do not allow to store parameters. Therefore, information might get lost when you save the data set in such a format. If in doubt, use the **amira** specific *AmiraMesh* format.

Chapter 4

Technical Information

This chapter contains technical information about **amira** which is not covered in the previous chapters.

- *Data Import*
- *Command Line Options*
- *Environment Variables*
- *amira start-up script*
- *System Requirements*

4.1 Data Import

Usually, one of the first things **amira** users want to know is how to import their own data into the system. This section contains some advice intended to ease this task.

In the simplest case, your data is already present in a standard file format supported by **amira**. To import such files, simply use the *File Load* menu. A *list of all supported formats* can be found in the index section of the user's guide. Usually, the system recognizes the format of a file automatically by analyzing the file header or the filename suffix. If a supported format is detected, the file browser indicates the format name.

Often, *3D image volumes* are stored slice by slice using standard 2D image formats such as TIFF or JPEG. In case of medical images, slices are commonly stored in ACR-NEMA or DICOM format. If you select multiple 2D slices simultaneously in the file browser, all slices will automatically be combined into a single 3D data set. Simultaneous selection is most easily achieved by first clicking the first slice and then shift-clicking the last one.

If your data is not already present in a standard file format supported by **amira** you will have to write your own converter or export filter. For many data objects such as 3D images, regular fields,

or tetrahedral grids **amira**'s native *AmiraMesh* format is most appropriate. Using this format you can even represent point sets or line segments for which there is hardly any other standard format. The *AmiraMesh documentation* explains the file syntax in detail and contains examples of how to encode different data objects. One important **amira** data type, triangular non-manifold surfaces, cannot be represented in a *AmiraMesh* file but has its own file format called *HxSurface* format.

Finally, in case of images or regular fields with uniform coordinates you may also read *binary raw data*. Note that for raw data the dimensions and the bounding box of the data volume must be entered manually in a dialog box which pops up after you have selected the file in the file browser.

4.2 Command Line Options

This section describes the command line options understood by **amira**. In general, on Unix systems **amira** is started via the `start` script located in the subdirectory `bin`. Usually, this script will be linked to `/usr/local/bin/amira` or something similar. Alternatively, the user may define an alias `amira` pointing to `bin/start`.

On Windows systems **amira** is usually started via the start menu or via a desktop icon. Nevertheless, the **amira** executable may also be invoked directly by calling `bin/arch-Win32-Optimize/amira.exe`. In this case, the same command line options as on a Unix system are understood.

The syntax of **amira** is as follows:

```
amira [options] [files ...]
```

Data files specified in the command line will be loaded automatically. In addition to data files, script files can also be specified. These scripts will be executed when the program starts.

The following options are supported:

- `-help`
Prints a short summary of command line options.
- `-version`
Prints the version string of **amira**.
- `-no_stencils`
Tells **amira** not to ask for a stencil buffer in its 3D graphics windows. This option can be set to exploit hardware acceleration on some low-end PC graphics boards.
- `-no_overlays`
Tells **amira** not to use overlay planes in its 3D graphics windows. Use this option if you experience problems when redirecting **amira** on a remote display.
- `-no_gui`
Starts up **amira** without opening any windows. This option is useful for executing a script in batch mode.

- `-logfile filename`
Causes any messages printed in the *console window* also to be written into the specified log file. Useful especially in conjunction with the `-no_gui` option.
- `-depth_size number`
This option is only supported on Linux systems. It specifies the preferred depth of the depth buffer. The default on Linux systems is 16 bits.
- `-style={windows | motif | cde}` This option sets the display style of **amira**'s Qt user interface.
- `-debug` This options applies to the developer version only. It causes local packages to be executed in debug version. By default, optimized code will be used.
- `-cmd command [-host hostname] [-port port]`
Send Tcl command to a running **amira** application. Optionally the host name and the port number can be specified. You must type `app -listen` in the console window of **amira** before commands can be received.

4.3 Environment Variables

In order to execute **amira** no special environment settings are required. On Unix systems some environment variables like the shared library path or the **amira** root directory are set automatically by the **amira** start script. Other environment variables may be set by the user in order to control certain features. These variables are listed below. On Unix systems environment variables can be set using the shell commands `setenv` (csh or tcsh) or `export` (sh, bash, or ksh). On Windows environment variables can be defined in the system properties dialog (Windows 2000/XP).

- **AMIRA_DATADIR**
A list of data directory names separated by semicolons [;] on Windows systems and colons [:] on Unix systems. The first directory will be used as the default directory of the file dialog. Other directories are quickly accessible via the file dialog's path list.
- **AMIRA_TEXMEM**
Specifies the amount of texture memory in megabytes. If this variable is not set some heuristics are applied to determine the amount of texture memory available on a system. However, these heuristics may not always yield a correct value. In such cases the performance of the *Voltex* module might be improved using this variable.
- **AMIRA_MULTISAMPLE**
On high-end graphics systems like SGI Onyx, a multi-sample visual is used by default. In this way, efficient scene anti-aliasing is achieved. If you want to disable this feature, set the environment variable **AMIRA_MULTISAMPLE** to 0. Note that on other systems, especially on PCs, anti-aliasing cannot be controlled by the application but has to be activated directly in the graphics driver.

- **AMIRA_NO_OVERLAYS**
If this variable is set, **amira** will not use overlay planes in its 3D graphics windows. The same effect can be obtained by means of the `-no_overlays` command line option. Turn off overlays if you experience problems with redirecting **amira** on a remote display, or if your X server does not support overlay visuals.
- **AMIRA_NO_SPLASH_SCREEN**
If this variable is set, **amira** will not display a splash screen while it is initializing.
- **AMIRA_LOCAL**
Specifies the location of the local **amira** directory containing user-defined modules. IO routines or modules defined in this directory replace the ones defined in the main **amira** directory. This environment variable overwrites the local **amira** directory set in the development wizard (see **amira** programmer's guide for details).
- **AMIRA_SMALLFONT**
Unix systems only. If this variable is set a small font will be used in all ports being displayed in the *Properties Area* even if the screen resolution is 1280x1024 or bigger. By default, the small font will be used only in case of smaller resolutions.
- **AMIRA_XSHM**
Unix systems only. Set this variable to 0 if you want to suppress the use of the X shared memory extension in **amira**'s *image segmentation editor*.
- **AMIRA_SPACEMOUSE**
This variable has to be set in order to support a spaceball or spacemouse device (see www.spacemouse.com). With the spacemouse you can navigate in the 3D viewer window. Two modes are supported, a rotate mode and a fly mode. You can switch between the two modes by pressing the spacemouse buttons 1 or 2.
- **AMIRA_STEREO_ON_DEFAULT**
If this variable is set the 3D viewer will be opened in OpenGL raw stereo mode by default. In this way some screen flicker can be avoided which otherwise occurs when switching from mono to stereo mode. Currently the variable is supported on Unix systems only.
- **TMPDIR**
This variable specifies in which directory temporary data should be stored. If not set, such data will be created under `/tmp`. Among others, this variable is interpreted by **amira**'s job queue.

4.4 User-defined start-up script

amira may be customized in certain ways by providing a user-defined start-up script. The default start-up script, called `Amira.init`, is located in the subdirectory `share/resources` of the **amira** installation directory. This script is read each time the program is started. Among other things, the start-up script is responsible for registering file formats, modules, and editors and for loading the default colormaps.

If a file called `Amira.init` is found in the current working directory, this file is read instead of the

default start-up script. If no such file is found, on Unix systems it is checked if there exists a start-up script called `.Amira` in the user's home directory. Below an example of a user-defined start-up script is shown:

```
# Execute the default start-up script
source $AMIRA_ROOT/share/resources/Amira.init 0

# Set up a uniform black background
viewer 0 setBackgroundMode 0
viewer 0 setBackgroundColor black

# Choose non-default font size for the help browser
help setFontSize 12

# Restore camera setting by hitting F2 key
proc onKeyF2 { } {
    viewer setCameraOrientation 1 0 0 3.14159
    viewer setCameraPosition 0 0 -2.50585
    viewer setCameraFocalDistance 2.50585
}
```

In this example, first the system's default start-up script is executed. This ensures that all **amira** objects are registered properly. Then some special settings are made. Finally, a hot-key procedure is defined for the function key F2. You can define such a procedure for any other function key as well. In addition, procedures like `onKeyShiftF2` or `onKeyCtrlF2` can be defined. These procedures are executed when a function key is pressed with the Shift or the Ctrl modifier key being pressed down.

4.5 System Requirements

amira runs on Microsoft Windows 2000/XP, Linux (Red Hat Enterprise Linux 3.0), Mac OS X Tiger (10.4), Sun Solaris 8 and 9, HP-UX 11.00, SGI Irix 6.5.x.

amira relies on fast hardware-accelerated OpenGL 3D graphics. We strongly recommend hardware texture mapping, since many visualization tools in **amira** rely on it. Hardware texture mapping is available, on all recent PC 3D graphics boards. On Unix systems it is available for example, on SGI O2, Octane and Onyx systems, on HP workstations with fx/4, fx/6 or fx/10 graphics, or on Sun Creator 3D, Elite, Expert 3D or newer graphics boards. For details on hardware acceleration, see below.

Apart from 3D graphics hardware probably, the most important system parameter is main memory. You should have at least 512 MB, preferably 2 GB or more. **amira** can also be started with less than 512 MB but it may decrease global performance.

The speed of the processor of course is also an important parameter. However it is less critical than

the graphics system and the main memory size. For the PC versions, we recommend at least a 2 GHz processor.

4.5.1 On System Stability

amira is a very demanding application that extensively uses high-end features. Experience shows that such applications tend to reveal instabilities in system hardware, hardware drivers, and the operating system. A common problem is insufficient main memory. We recommend you configure enough swap memory in addition to physical memory. The total amount of virtual memory should be at least 1 GB. 2 GB would be even better.

Especially on PC platforms, OpenGL drivers today are often not as robust as desired. Also, system crashes due to bad memory chips or unstable power-supply are not rare. If you experience problems or instabilities with **amira** on your Windows platform, we recommend that you follow these steps:

1. Click on all the demo scripts in the Online User's Guide. If the system crashes, turn off hardware acceleration (choose the *extended* button from the Windows display settings dialog) and try again. If this eliminates the problem, there is a bug in your OpenGL driver. Try to get a new driver from the web site of the manufacturer of your graphics board.
2. Try using a different color depth in the Windows display settings dialog. Try 24 or 32 bit.
3. Load the `lobus.am` data set and visualize it with a Voltex module. Turn on the spin rotation (turn it with the mouse in the viewer and release the mouse button while moving the mouse, so that the object continues moving). Let it run overnight (turn off the screen saver). If the system has crashed or frozen the next morning, you probably have a hardware problem.

If this does not help, or if a reproducible error occurs on different computers, then it might be a bug in the **amira** software itself. Please report such bugs so that they can be eliminated in the next release or a patch can be prepared.

4.5.2 Microsoft Windows

amira runs on Intel or AMD-based systems with Microsoft Windows 2000 and Microsoft Windows XP.

Graphics Hardware: You should use a graphics board with OpenGL support and texture mapping capabilities. Both is the case for almost all newer 3D boards.

4.5.3 Silicon Graphics

Graphics Hardware: To get optimal graphics performance, the machine should support *texture mapping in hardware*. Currently this is the case for all O2 systems, and for Octane systems with High Impact, Maximum Impact (not Solid Impact) and Odyssey graphics. **amira** provides a number of modules which make use of texture mapping, e.g., slicing, pseudo-coloring, or volume rendering. On

machines without hardware texture mapping, these modules either run much slower or may not work at all. The advantage of the Octane is a higher speed in polygon rendering. For a complex model with an isosurface of 100,000 triangles, the frame rate is 10 per second for an Octane, compared to 3 per second for an O2. The MXE and MXI Octanes have larger texture memory than the SSE and SSI. Thus the MXE and MXI enable a direct volume rendering using 3D textures, which is not possible on an SE, SI or O2.

Software: The current version of **amira** requires IRIX 6.5.x or higher. We recommend you install the newest version of the operating system (see <http://www.sgi.com/support>).

4.5.4 HP-UX

amira performs pretty well on HP workstations equipped with Visualize fx/4, fx/6+, and fx/10 graphics cards under HP-UX 11.00. Probably it will run on other machines as well provided the OpenGL runtime environment has been installed. In any case we recommend to install the texture acceleration option for your graphics system (hardware texture mapping), especially if you intend to work with large 3D image data sets.

Important: By default some HP workstations are configured with a data size limit of 64 MB for each process. In order to load reasonable data sets, you should increase this value to 1 GB and the stack size to 128 MB. Do this by modifying the values in `sam/Kernel Configuration` `maxdsiz=0x80000000`, `ssiz=0x80000000`, `tssiz=0x80000000`.

4.5.5 SunOS

amira runs on Sun workstations with Solaris 8 or Solaris 9.

amira is successfully being used on systems with Creator 3D, Elite 3D, Expert 3D, and Zulu graphics boards. It runs on a simple Creator graphics boards as well. However, since no hardware texturing is available, performance is limited.

4.5.6 Linux

The Linux version of **amira** as been developed and tested on Red Hat Enterprise Linux 3.0. On other Linux distributions this version might not run because certain required system libraries are missing or because different versions of these libraries are installed.

There is also a version for Linux IA64 or AMD64 systems. This version has been compiled and tested on the Red Hat Enterprise Linux 3.0 distribution.

amira works with the current 3D graphics drivers from nVidia and ATI under XFree86 4/Xorg. It has also been successfully tested with other X-servers like the *3D Accelerated-X* servers from XI graphics.

Note: After a standard installation of Red Hat Enterprise 3.0 Linux, hardware acceleration is not necessarily activated, although X-Windows and **amira** may work fine. To enable OpenGL

hardware acceleration specific drivers may have to be installed, like the nVidia drivers from <http://www.nvidia.com>. This can increase rendering performance by an order of magnitude. Sometimes it is necessary to disable the stencil buffers (by starting **amira** with the option `-no_stencils`) to get acceleration.

4.5.7 Mac

The Mac version of **amira** has been developed and tested on Mac OS X Tiger (10.4). On other later Mac releases this version might not run because certain required system libraries may be different.

amira works with the current Mac 3D graphics drivers from nVidia and ATI.

In any case **amira** requires an X server resolution of at least 1024x768 and at least 15 bits of color depth. We strongly recommend 1280x1024 with 24-bit color depth at least.

Please note that if software rendering is used, rendering performance may drop significantly, especially for visualization techniques like volume rendering.

4.6 **amira** and the /3GB switch

This page describes problems and possible solutions related to the usage of large data sets in **amira** 4.1 on computers running Windows XP. If you frequently encounter dialogs in **amira** such as "cannot allocate xxxxxx bytes of memory" although you think you have enough physical memory installed, then the information provided on this page might be useful for you.

The Problem:

Any 32-bit operating system such as Windows (NT4.0, 2000, XP) or Linux can manage at most 4 gigabytes (GB) of memory. Windows divides this addressable space into 2 GB reserved for usage by the system only and 2 GB for any user application. Therefore, a single application can use at maximum 2 GB of memory. Note that this is independent of the actual physical memory, i.e., it does not matter whether the system has 1, 2, 3 or 4 GB of memory installed. Within the 2 GB reserved for the user, the software itself (executable and all of its DLLs) and all of the data has to fit. This makes clear that it is impossible for a Windows program to load 2 GB of data at a time.

To overcome this, Microsoft has provided a boot option for Windows that alters partitioning of address space so that 3 GB are reserved for the user and 1 GB for the system. Unfortunately, however, this boot option is not compliant with most Windows XP installations at level SP1. A description of the problem is given here. Microsoft has fixed the problem with its current Service Pack 2.

How to activate the /3GB switch:

It should be noted that we cannot guarantee that the following changes to be made in your system will not affect the execution of other programs although we have not found any incompatibilities so far. We also wish to stress that we assume no responsibility for any loss of data as a consequence of those changes. *We therefore strongly recommend that you back up your system before making these changes.*

All you need to do is to edit the file C:/boot.ini. To do so, make sure that you are logged on your machine with administrative privileges. Then select from Start->Settings->Control Panel->System->Advanced->(Startup and Recovery) Settings and click the Edit button. Duplicate the line under [operating systems]. Change the name between the quotes and add the "/3GB" switch right after the "/fastdetect" switch. The new entry might look then similar to this:

```
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP \
Professional" /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP \
Professional 3GB" /fastdetect /3GB
```

Note that for the usage of the /3GB boot option the application must be aware of the altered memory assignment. For *amira* 4.1 there is already native support for the /3GB boot option. If you are using an older version of *amira*, we strongly recommend updating to 4.1.

Now, as you boot up your computer, a screen is displayed that gives you the choice between two boot options. Use the arrow keys to select the entry with the "3GB" in the line and press return. *amira* should now be able to address 3 GB of memory instead of the default 2 GB. You can appreciate this by comparing the result of "app maxmalloc" in the *amira* console window with and without the 3 GB option. On a typical installation with 1 GB of memory installed and without the /3GB switch, the command "app maxmalloc" may result in something like this:

```
Allocated chunk of 942 MB.
Allocated chunk of 225 MB.
Allocated chunk of 129 MB.
Allocated chunk of 121 MB.
Allocated chunk of 72 MB.
Allocated a total of 1723 MB in 25 chunks
```

With the /3GB switch on the same installation, the command results in the following:

```
Allocated chunk of 1023 MB.
Allocated chunk of 919 MB.
Allocated chunk of 225 MB.
Allocated chunk of 111 MB.
Allocated a total of 2278 MB in 4 chunks
```

What you can see is that the total amount of allocatable memory has not changed dramatically. However, with the /3GB switch, the second largest chunk of memory is nearly as large as the first one.

Chapter 5

Scripting

5.1 Introduction

This chapter is intended for advanced **amira** users only. If you do not know what scripting is, it is very likely that you will not need the features described in this chapter.

Beside the interactive control via the graphical user interface, most of the **amira** functionality can also be accessed using specific commands. This allows you to automate certain processes and to create scripts for managing routine tasks or for presenting demos. **amira**'s scripting commands are based on Tcl, the *Tool Command Language*. This means you can write command scripts using Tcl with **amira**-specific extensions.

amira commands can be typed into the **amira** console window, as described in Section 3.1.9. Commands typed directly into the console window will be executed immediately. Alternatively, commands can be written into a text file, which can then be executed as a whole.

This chapter is organized as follows:

Section 5.2 (Introduction to Tcl) gives a short introduction to the Tcl scripting language. This section is not very **amira** specific.

Section 5.3 (**amira** Script Interface) explains **amira**-specific commands and concepts related to scripting. This includes a *reference of global commands*.

Section 5.4 (**amira** Script Files) explains the different ways of writing and executing script files including references to script objects, resource files, and function-key bound Tcl procedures.

Section 5.5 (Configuring Popup Menus) describes how the popup menu of an object can be configured using script commands, and how new entries causing a script to be executed can be created.

Section 5.6 (Registering pick callbacks) describes how script callbacks can be attached to objects or viewers and be invoked on user pick events.

Data Type: Script Object describes how to use Tcl scripts for defining custom modules that have their own graphical user interface and can be used like the built-in **amira** objects.

5.2 Introduction to Tcl

This chapter gives a brief introduction to the Tcl scripting language. If you are familiar with Tcl you can skip this section. However, please notice that instead of the `puts` command, you should use `echo` for output to the **amira** console.

This chapter is not intended to cover all details of the language. For a complete documentation or reference manual of the Tcl language, refer to a text book like *Tcl and the Tk Toolkit* by John K. Ousterhout, the creator of Tcl. Like many other books about Tcl, this also covers the Tk GUI toolkit. Note that Tk is not used in **amira**.

Alternatively you can easily find Tcl documentation and reference manuals on the internet e.g., at www.scriptics.com, or looking up keywords like `Tcl tutorial` or `Tcl documentation` with a search engine like www.google.com.

When you type Tcl commands into the **amira** console, they will be executed as soon as the return key is pressed. Use the completion and history functions provided by the **amira** console, as described in Section 3.1.9 (console window).

5.2.1 Tcl Lists, Commands, Comments

First, please note that Tcl is case sensitive: `set` and `Set` are not the same.

A Tcl command is a space-separated list of words. The first word represents the command name, all further words are treated as arguments to that command. As an example, try the **amira**-specific command `echo`, which will print all its arguments to the **amira** console. Try typing

```
echo Hello World
```

This will output the string *Hello World*. Note that Tcl commands can be separated by a semi-colon (;) or a newline character. If you want to execute two successive `echo` commands, you can do it this way:

```
echo Hello World ; echo Hello World2
```

or like this:

```
echo Hello World
echo Hello World2
```

Instead of a command, you can also place a comment in Tcl code. A comment starts with a hash character (#) and is ended by the next line break:

```
# this is a comment
echo Hello World
```

5.2.2 Tcl Variables

In Tcl, variables can be used. A variable represents a certain state or value. Using Tcl code, the value of the placeholder can be queried, defined, and modified. To define a variable use the command

```
set name value
```

e.g.

```
set i 1
set myVar foobar
```

Note that in Tcl internally all variables are of string type. Since the set command requires exactly one argument as the variable value, you have to quote values that contain spaces:

```
set Output "Hello World"
```

or

```
set Output {Hello World}
```

In order to substitute the value of a variable with name *varname*, a \$ sign has to be put in front of that name. The expression \$varname will be replaced by the value of the variable. After the above definitions,

```
echo $Output
```

would print

```
Hello World
```

in the console window, and

```
echo "$i.) $Output"
```

would yield the output *1.) Hello World*. Note that variable substitution is performed for strings quoted in ", while it is not done for strings enclosed in braces {}. Even newline characters are allowed in a {} enclosed string. Note however that it is not possible to type in multi-line commands into the amira console.

5.2.3 Tcl Command Substitution

To do mathematical computations in Tcl, you can use the command `expr` which will evaluate its arguments and return the value of the expression. Examples are:

```
expr 5 / ( 7 + 3 )
expr $i + 1
```

In order to use the result of a command like `expr` for further commands, an important Tcl mechanism has to be used: command substitution, denoted by brackets `[]`. Any list enclosed in brackets `[]` will be executed as a separate command first, and the `[...]` construct will be replaced with the *result* of the command. This is similar to the `'...'` construct in Unix command shells. For example, in order to increase the value of the variable `i` by one you can use:

```
set i [expr $i + 1]
```

Of course, command expressions can be arbitrarily nested. The order of execution is always from the innermost bracket pair to the outermost one:

```
echo [expr 5 * [expr 7 + [expr 3+3]]]
```

5.2.4 Tcl Control Structures

Further important language elements are `if-else` constructs, `for` loops and `while` loops. These constructs typically are multi-line constructs and therefore can not be typed conveniently into the `amira` console. If you want to try the examples shown below, write them into a file like `C:\test.txt` by using a text editor of your choice, and execute the file by typing

```
source C:\test.txt
```

We start with the `if-then` mechanism. It is used to execute some code *conditionally*, only if a certain *expression* evaluates to “true (meaning a value different from 0):

```
set a 7
set b 8
if {$a < $b} {
    echo "$a is smaller than $b"
} elseif {$a == $b} {
    echo "$a equals $b"
} else {
    echo "$a is greater than $b"
}
```

The `elseif` and `else` parts are optional. Multiple `elseif` parts can be used, but only a single `if` and `else` part.

Another important construct is the conditional loop. Like the `if` command, it is based on checking a conditional expression. In contrast to `if`, the conditional code is executed multiple times, as long as the expression evaluates to true:

```
for {set i 1} {$i < 100} {set i [expr $i*2]} {  
    echo $i  
}
```

In fact this code is identical to:

```
set i 1  
while {$i < $100} {  
    echo $i  
    set $i [expr $i * 2]  
}
```

Both loops would produce the output *1, 2, 4, 8, 16, 32, 64*.

If you want to execute a loop for all elements of a list, there is another very convenient command for that:

```
foreach x {1 2 4 8 16 32 64} {  
    echo $x  
}
```

This will generate the same output as the previous example. Note that the expression enclosed in braces is a space-separated list of words.

User-Defined Tcl Procedures

A new function or procedure is defined in Tcl using the `proc` command. `Proc` takes two arguments: a list of argument names and the Tcl code to be executed. Once a procedure is defined, it can be used just like any other Tcl command:

```
proc computeAverageA {a b} {  
    return [expr ($a+$b)/2.0]  
}  
proc computeAverageB {a b c} {  
    return [expr ($a+$b+$c)/3.0]  
}
```

```
echo "average of 2 and 3: [computeAverageA 2 3]"
echo "average of 2,3,4: [computeAverageB 2 3 4]"
```

As you can see in the example, the argument list defines the names for local variables that can be used in the body of the procedure (e.g. `$a`). The `return` command is used to define the result of the procedure. This result is the value that is used in the command bracket substitution `[]`.

If you want to define a procedure with a flexible number of arguments, you must use the special argument name `args`. If the argument list contains just this word, the newly defined command will accept an arbitrary number of arguments, and these arguments are passed as a *list* called `args`:

```
proc computeAverage args {
    set result 0
    foreach x $args {
        set result [expr $result + $x]
    }
    return [expr $result / [llength $args]]
}
```

In this example, the `llength` command returns the number of elements contained in the `args` list.

Note that the variable `result` defined in the procedure has *local* scope, meaning that it will not be known outside the body of the procedure. Also, the value of globally defined variables is not known within a procedure unless that global variable is declared using the keyword `global`:

```
set x 3
proc printX {} {
    global x
    echo "the value of x is $x"
}
```

There is much more to be said about procedures, e.g., concerning argument passing, evaluation of commands in the context outside of the procedure, and so on. Please refer to a Tcl reference book for these advanced topics.

List and String Manipulation

Finally, at the end of this brief Tcl introduction, we come back to the concept of lists. Basically everything in Tcl is constructed using lists, so it is very important to know the most important list manipulation commands as well as to understand some subtle details.

Here is an example of how to take an input list of numbers and construct an output list in which each element is twice as big as the corresponding element in the input list:


```

set input [list 1 2 3 4 5]
set output [list]
foreach element $input {
    lappend output [expr $element * 2]
}

```

You can think of lists as simple strings in which the list elements are separated by spaces. This means that you can achieve the same result as in the previous example without using the list commands:

```

set input "1 2 3 4 5"
set output ""
foreach element $input {
    append output "[expr $element * 2] "
}

```

The *append* command is similar to *lappend*, but it just adds a string at the end of an existing string. List manipulation becomes much more involved when you start nesting lists. Nested lists are represented using nested pairs of braces, e.g.

```

set input {1 2 {3 4 5 {6 7} 8 } 9}
foreach x $input {
    echo $x
}

```

The result of this command will be

```

1
2
3 4 5 {6 7} 8
9

```

Please note that Tcl will automatically *quote* strings that are not single words when constructing a list. Here is an example:

```

set i [list 1 2 3]
lappend i "4 5 6"
echo $i

```

will yield the output

```

1 2 3 {4 5 6}

```

You can use the *lindex* command to access a single element of a list. *lindex* takes two arguments: the list and the index number of the desired element, starting with 0:

```
set i [list a b c d e]
echo [lindex $i 2]
```

will yield the result *c*.

5.3 amira Script Interface

Although the Tcl language is not intrinsically object oriented, the **amira** script interface is. There is one command for each object in the **amira** Pool. In addition there are several *global commands* associated with global objects in **amira** such as the *viewer* or the **amira** .

A command associated with an object in the Pool (e.g., an OrthoSlice module or an Isosurface module) only exists while the object exists. These commands are identical to the name of the object as displayed in the Pool. Typically the script interface of a specific object contains many different functions. The general syntax for an **amira** object-related command is

```
<object-name> <command-word> <optional-arguments> ...
```

For example, if an object called GlobalAxis exists (choose View/Axis from the **amira** menu) then you can use commands like

```
GlobalAxis deselect
GlobalAxis select
GlobalAxis setIconPosition 100 100
```

Remember to use the completion and history functions provided by the **amira** console, as described in Section 3.1.9 (console window) to save typing.

If you have already used **amira** you have noticed that the parameters and the behavior of an **amira** module are controlled via its *ports*. The ports provide a user interface to change their values when the module is selected. All ports can also be controlled via the command interface. The general syntax for that is

```
<object-name> <port-name> <port-command> <optional-arguments> ...
```

For example for the GlobalAxis you can type

```
GlobalAxis options setValue 1 1
GlobalAxis thickness setValue 1.5
GlobalAxis fire
```

When you type in these commands you will notice that the values in the user interface change immediately. However the module's compute method is not called until explicitly firing the module using the `fire` command. This allows you to first set values for multiple ports without a recomputation after each command. However, note that some modules automatically reset some of their ports for example when a new input object is connected. In such cases you may need to call `fire` after setting the value of every single port.

Usually the name of a port is identical to the text label displayed in the graphical user interface, except that white spaces are removed and command names start with a lower case letter. To find out the names of all ports of a specific module use the command

```
<object-name> allPorts
```

Almost all ports provide a `setValue` and a `getValue` command. The number of parameters and the syntax, of course, depend on the ports.

Commands of the type `<object-name> <port-name> setValue ...` make up more than 90% of a typical **amira** script. However, besides the port commands, many **amira** objects provide additional specific commands. The command interface for a particular module is described in the User's Reference Guide. You can quickly find the corresponding page by clicking the ? button in the Properties Area when the module has been selected.

As a quick help, entering an object's name without further options will display all commands available for that object. Note that this will also show undocumented, unreleased, and experimental commands. In order to get more information about a particular module or port command, you can type it into the console window without any arguments and then press the F1 key. This opens the help browser with a command description.

amira objects are part of a class hierarchy. Similar to the C++ programming interface, also script commands are inherited by derived classes from its base classes. This means that a particular object like the axis object will beside its own specific commands also provide all the commands available in its base classes. Links to the base class commands are given in a module's documentation.

5.3.1 Predefined Variables

There exist some variables in **amira** Tcl, which are predefined and have a special meaning. These are

- `AMIRA_ROOT`: **amira** installation directory.
- `AMIRA_LOCAL`: Personal **amira** development directory (Developer Pack only).
- `SCRIPTFILE`: Tcl script file currently executed.
- `SCRIPTDIR`: Directory in which currently executed script resides.
- `hideNewModules`: If set to 1, icons of newly created modules will initially be hidden.

5.3.2 Object commands

The basic command interface of **amira** modules and data objects is described in the data type chapter of the reference part of the user's guide in the *Object* section. The basic syntax of object commands is

```
<object> <command> <arguments> ...
```

where `<object>` refers to the name of the object and `<command>` denotes the command to be executed. Each module or data object may define its own set of commands in addition to the commands defined by its base classes. The commands described in the *Object* section are provided by all modules and data objects.

In the following section *Global commands* are described.

5.3.3 Global commands

This section lists **amira**-specific global Tcl commands. Some of these commands are associated with certain global objects in **amira**, such as the console window, the main window, or the viewer window. Other commands are such as `load` or `echo` are not. These commands are described in one common subsection. In summary, the following command sections are provided:

- *viewer command options* (`viewer`)
- *main window command options* (`theMain`)
- *console command options* (`theMsg`)
- *common commands for top-level windows*
- *progress bar command options* (`workArea`)
- *application command options* (`app`)
- *other global commands*

5.3.3.1 Viewer command options

Commands to a viewer can be entered in the console window. The syntax is

```
viewer [<number>] command,
```

where `<number>` specifies the viewer being addressed. The value 0 refers to the main viewer and may be omitted for convenience.

Commands

```
viewer [<number>] snapshot [-offscreen [<width> <height>]]  
<filename>
```

This command takes a snapshot of the current scene and saves it under the specific filename.

The image format will be automatically determined by the extension of the file name. The list of available formats includes: TIFF (.tif, .tiff), SGI-RGB (.rgb, .sgi, .bw), JPEG (.jpg, .jpeg), PNM (.pgm, .ppm), BMP (.bmp), PNG (.png), and Encapsulated PostScript (.eps). If the viewer number is not given, the snapshot is taken from all viewers, if you have selected the 2 or 4 viewer layout from the *View menu*.

If the `-offscreen` option is specified, offscreen rendering with a maximum size of 2048x2048 is used. In this case the viewer number is required even if viewer 0 is addressed. If the width and height is not specified explicitly, the size of the image is the current size of the viewer.

Caution: If you have more than one transparent object visible in the viewer and you want to use offscreen rendering set the transparency mode to *Blend Delayed* and check to see if all objects are rendered properly prior to taking a snapshot.

```
viewer [<number>] setPosition <x> <y>
```

Sets the position of the viewer window relative to the upper left corner of the screen. If more than one viewer is shown in the same window the position of the toplevel window is set.

```
viewer [<number>] getPosition
```

Returns the position of the viewer window. If more than one viewer is shown in the same window the position of the toplevel window is returned.

```
viewer [<number>] setSize <width> <height>
```

Sets the size of the viewer window. Width and height specify the size of the actual graphics area. The window size might be a little bit larger because of the viewer decoration and the window frame.

```
viewer [<number>] getSize
```

Returns the size of the viewer window without decoration and window frame.

```
viewer [<number>] setCamera <camera-string>
```

Restores all camera settings. The camera string should be the output of a `getCamera` command.

```
viewer [<number>] getCamera
```

This command returns the current camera settings, i.e., position, orientation, focal distance, type, and height angle (for perspective cameras) or height (for orthographic cameras). The values are returned as *amira* commands, which can be executed in order to restore the camera settings. The complete command string may also be passed to `setCamera` at once.

```
viewer [<number>] setCameraPosition <x> <y> <z>
```

Defines the position of the camera in world coordinates.

```
viewer [<number>] setCameraPosition <x> <y> <z>
```

Returns the position of the camera in world coordinates.

```
viewer [<number>] setCameraOrientation <x> <y> <z> <a>
```

Defines the orientation of the camera. By default, the camera looks in negative z-direction with the

y-axis pointing upwards. Any other orientation may be specified as a rotation relative to the default direction. The rotation is specified by a rotation axis $x\ y\ z$ followed by a rotation angle a (in radians).

`viewer [<number>] getCameraOrientation`

Returns the current orientation of the camera in the same format used by `setCameraOrientation`.

`viewer [<number>] setCameraFocalDistance <value>`

Defines the camera's focal distance. The focal distance is used to compute the center around which the scene is rotated in interactive viewing mode.

`viewer [<number>] getCameraFocalDistance`

Returns the current focal distance of the camera.

`viewer [<number>] setCameraHeightAngle <degrees>`

Sets the height angle of a perspective camera in degrees. The smaller the angle the bigger the field of view. The command has no effect if the current camera is an orthographic one.

`viewer [<number>] getCameraHeightAngle`

Returns the height angle of a perspective camera.

`viewer [<number>] setCameraHeight <height>`

Sets the height of the view volume of an orthographic camera. The command has no effect if the camera is an perspective one.

`viewer [<number>] getCameraHeight`

Returns the height of an orthographic camera.

`viewer [<number>] setCameraType <perspective|orthographic>`

Sets the camera type.

`viewer [<number>] getCameraType`

Returns the camera type.

`viewer [<number>] setTransparencyType <type>`

This command defines the strategy used for rendering transparent objects. The argument *type* may be a number between 0 and 6, corresponding to the entries *Screen Door*, *Add*, *Add Delay*, *Add Sorted*, *Blend*, *Blend Delay*, and *Blend Sorted* as described for the *View menu*.

Most accurate results are obtained using mode 6, which is the default. However, some objects may not be recognized correctly as being transparent. In this case you may switch them off and on again in order to force them to be rendered last. Also, if lines are to be rendered on a transparent background problems may occur. In this case, you may use transparency mode 4 and ensure the correct rendering order manually.

`viewer [<number>] getTransparencyType`

This command returns the current transparency type as a number in the range 0...6. The meaning of this number is the same as in `setTransparencyType`.

```
viewer [<number>] setBackgroundColor <r> <g> <b>
```

This command sets the color of the background to a specific value. The color may be specified either as a triple of integer RGB values in the range 0...255, as a triple of rational RGB values in the range 0.0...1.0, or simply as plain text, e.g., *white*, where the list of allowed color names is defined in `/usr/lib/X11/rgb.txt`.

```
viewer [<number>] getBackgroundColor
```

Returns the primary background color as an RGB triple with values between 0 and 1.

```
viewer [<number>] setBackgroundColor2 <r> <g> <b>
```

Sets the secondary background color which is used by non-uniform background modes.

```
viewer [<number>] getBackgroundColor2
```

Returns the secondary background color as an RGB triple with values between 0 and 1.

```
viewer setBackgroundMode <mode>
```

Allows you to specify different background patterns. If mode is set to 0 a uniform background will be displayed. Mode 1 denotes a gradient background. Mode 2 causes a checkerboard pattern to be displayed. This might help to understand the shape of transparent objects. Finally, mode 3 draws an image previously defined with `setBackgroundImage` on the background.

```
viewer getBackgroundMode
```

Returns the current background mode.

```
viewer setBackgroundImage <imagefile> [<imagefile2>] [-stereo]
```

This command allows you to place an arbitrary raster image into the center of the viewer's background. The image must not be larger than the viewer window itself. Otherwise it will be clipped. The format of the image file will be detected automatically by looking at the file name extension. All formats mentioned for the `snapshot` command are supported except of Encapsulated PostScript. If a second image file is specified, this file will be used as the right eye image in case of active stereo rendering. If the options `-stereo` is specified and only one image file is given, it is assumed that this file contains a left eye view and a right eye view composited side by side. These views then will be separated automatically.

```
viewer getBackgroundImage
```

This command returns the file name of the last background image file defined with `setBackgroundImage`. If a pair of stereo images was specified, two file names are returned. If the option `-stereo` was used in `setBackgroundImage`, this option will be returned too.

```
viewer [<number>] setAutoRedraw <state>
```

If *state* is 0, the auto redraw mode is switched off. In this case the image displayed in the viewer window will not be updated, unless a redraw command is sent. If *state* is 1, the auto redraw mode is switched on again. In a script it might be useful to disable the auto redraw mode temporarily.

```
viewer [<number>] isAutoRedraw
```

Returns true if auto redraw mode is on.

`viewer [<number>] redraw`

This command forces the current scene to be redrawn. An explicit *redraw* is only necessary if the auto redraw mode has been disabled.

`viewer [<number>] rotate <degrees> [x|y|z|m|u|v]`

Rotates the camera around an axis. The axis to be taken is specified by the second argument. The following choices are available:

- x: the x-axis (1,0,0)
- y: the y-axis (0,1,0)
- z: the z-axis (0,0,1)
- m: the most vertical axis of x, y, or z
- u: the viewer's up direction
- v: the view direction

`amira service`

The last option does the same as the rotate button of the user interface. In most cases the *m* option is most adequate. For backward-compatibility the default is *u*.

`viewer [<number>] setDecoration <state>`

The decoration is an extended window frame that serves as a user-interface. It contains buttons and thumb wheels for adjusting the view or switching between interaction and viewing mode. The *decoration* command can be used to show or hide the decoration area. Hiding the decoration is useful when multiple viewers are open and the size of a single viewing window is rather small.

`viewer [<number>] saveScene <filename>`

Saves all of the geometry displayed in a viewer in Open Inventor 3D graphics format. Warning: Since many *amira* modules use custom Open Inventor nodes, the scene usually can not be displayed correctly in external programs like *ivview*.

`viewer [<number>] viewAll`

Resets the camera so that the whole scene becomes visible. This method is called automatically for the first object being displayed in a viewer.

`viewer [<number>] show`

This command opens the specified viewer and ensures that the viewer window is displayed on top of all other windows on the screen.

`viewer [<number>] hide`

This command closes the specified viewer.

`viewer [<number>] fogRange <min> <max>`

Sets a range of attenuation for the fog affect that can be introduced into a viewer scene by the *View menu*. The default range is [0, 1]. Values within this range correspond to distances of scene points

from the camera, such that points nearest to the camera have value zero and those farthest away have value one. Restricting the range of attenuation means that attenuation will start at points where the specified minimum is attained and reach its maximum at points where the specified maximum is attained. Maximum attenuation by fog is equivalent to invisibility, thus all points beyond that maximum will appear as background.

```
viewer [<number>] setVideoFormat pal|ntsc
```

Sets the size of the viewer window according to PAL 601 or NTSC 601 resolution, i.e., 720x576 pixels or 720x486 pixels. The current setting of the decoration is taken into account.

```
viewer [<number>] setVideoFrame <state>
```

If *state* is 1, a frame is displayed in the overlay plane of the viewer. This frame depicts the area where images recorded to video are safely shown on video players. Setting *state* to 0 switches the frame off. Note: Objects displayed in the overlay planes are not saved to file with the snapshot command (see above).

5.3.3.2 Main window command options

The command `theMain` allows you to access and control the **amira** main window. Besides the specific command options listed below also all sub-commands listed in Section 5.3.3.4 (Common commands for top-level windows) can be used.

Commands

```
theMain snapshot filename
```

Creates and saves a snapshot image of the main window. The format of the image file is determined from the file name extension. Any standard image file format supported by **amira** can be used, e.g., .jpg, .tif, .png, or .bmp.

```
theMain setViewerTogglesOnIcons {0|1}
```

Enables or disables the display of the orange viewer toggles on object icons in the **amira** Pool.

```
theMain ignoreShow [0|1]
```

Enables or disables the special purpose *no show flag*. If this flag is set, subsequent `mainWindow show` commands are ignored. This can be useful to run standard **amira** scripts in a VR Pack environment. Calling the command without an argument just returns the current value of the flag.

5.3.3.3 Console command options

The command `theMsg` allows you to access and control the **amira** console window. Besides the specific command options listed below also all sub-commands listed in Section 5.3.3.4 (Common commands for top-level windows) can be used.

Commands

```
theMsg error <message> [<btn0-text>] [<btn1-text>]  
[<btn2-text>]
```

Pops up an error dialog with the specified message. The dialog can be configured with up to three different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

```
theMsg warning <message> [<btn0-text>] [<btn1-text>]  
[<btn2-text>]
```

Pops up a warning dialog with the specified message. The dialog can be configured with up to three different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

```
theMsg question <message> [<btn0-text>] [<btn1-text>]  
[<btn2-text>]
```

Pops up a question dialog with the specified message. The dialog can be configured with up to three different buttons. The command blocks until the user presses a button. The id of the pressed button is returned.

```
theMsg overwrite <filename>
```

Pops up a dialog asking the user if it is ok to overwrite the specified file. If the user clicks *Ok*, 1 is returned, otherwise 0.

5.3.3.4 Common commands for top-level windows

These commands are available for all *amira* objects which open a separate top-level window. In particular, these are the *amira* main window (*theMain*), the console window (*theMsg*), and the viewer window (*viewer 0*). For example, you can set or get the position of these windows using the corresponding global command followed by *setPosition* or *getPosition*.

Commands

```
getFrameGeometry
```

Returns the position and size of the window including the window frame. In total four numbers are returned. The first two numbers indicate the position of the upper left corner of the window frame relative to the upper left corner of the desktop. The last two numbers indicate the window size in pixels.

```
getGeometry
```

Returns the position and size of the window without the window frame. In total four numbers are returned. The first two numbers indicate the position of the upper left corner of the window relative to the upper left corner of the desktop. The last two numbers indicate the window size in pixels.

`getPosition`

Returns the position of the upper left corner of the window including the window frame. This is the same as the first two numbers returned by `getFrameGeometry`.

`getRelativeGeometry`

Returns the position and size of the window including the window frame in relative coordinates. The size of the desktop is (1,1). The position and size of the window is specified by fractional numbers between 0 and 1.

`getSize`

Returns the size of the window without the window frame. This is the same as the last two numbers returned by `getGeoemtry`.

`hide`

Hides the window.

`setCaption <text>`

Sets the window title displayed in the window frame.

`setFrameGeometry <x y width height>`

Sets the position and size of the window including the window frame. Four numbers need to be specified, the x- and y-positions, the window width and the window height.

`setGeometry <x y width height>`

Sets the position and size of the window without the window frame. Four numbers need to be specified, the x- and y-positions, the window width and the window height.

`setPosition <x y>`

Sets the position of the upper left corner of the window frame.

`setRelativeGeometry <x y width height>`

Sets the position and size of the window including the window frame in relative coordinates. The size of the desktop is (1,1). The position and size of the window is specified by fractional numbers between 0 and 1.

`setSize <width height>`

Sets the size of the window without the window frame.

`show`

Makes the window visible in normal state. Also raises the window.

`showMinimized`

Makes the window visible in iconified state.

`showMaximized`

Makes the window visible in maximized state.

5.3.3.5 Progress bar command options

The command `workArea` allows you to access the progress bar located in the lower part of the `amira` main window. You can print messages or check if the stop button was pressed.

Commands

`workArea setProgressInfo <text>`

Sets an info text to be displayed in the progress bar. The text can be used to describe the status during some computation.

`workArea setProgressValue <value>`

Sets the value of the progress bar. The argument must be a floating point number between 0 and 1. For example, a value of 0.8 indicates that 80% of the current task has been done.

`workArea startWorking [<message>]`

Activates the stop button. After calling this command the `amira` stop button becomes active. In your script you can check if the stop button was hit by calling `workArea wasInterrupted`. When the stop button is active you can't interact with any other widget unless you call `workArea stopWorking` in your script. Therefore you must not enter this command directly in the console window, but you should only use it in a script file or in a Tcl procedure.

`workArea stopWorking`

Deactivates the stop button. Call this command when the compute task started with `workArea startWorking` is done or if the user pressed the stop button. This command also restores the progress info text which was shown before calling `startWorking`.

`workArea wasInterrupted`

Checks if the user pressed the stop button. You should only use this command between `workArea startWorking` and `workArea stopWorking`. If there are multiple nested compute tasks and the user presses the stop button, all subsequent calls to `wasInterrupted` return true until the first level is reached.

5.3.3.6 Application command options

The `app` command provides several options not related to a particular object or component in `amira`, but related to `amira` itself.

Commands

`app version`

Returns the current `amira` version.

`app uname`

Returns simplified name of operating system.

`app arch`

Returns **amira** architecture string, e.g., arch-Win32-Optimize, arch-IRIX64-Optimize.

`app hostid`

Returns host id needed to create an **amira** license key.

`app listen [port]`

Opens a socket to which Tcl commands can be sent. The TCP/IP port can be specified optionally. **WARNING:** This can create security holes. Do not use this unless behind a firewall and if you know what you are doing.

`app close`

Closes the **amira** Tcl port.

`app port`

Returns port number of **amira** Tcl port. Returns -1 if socket has not been opened.

`app send <command> [<host> [<port>]]`

Sends a Tcl command to a listening **amira**. If no host or port are specified, the **amira** instance will send the command to itself.

`app opengl`

Retrieve information about the used OpenGL driver including version number and supported extensions. This is useful information to send to the hotline if reporting rendering problems.

`app cluster`

Returns the current node status which can be "master" or "slave" if some cluster mode is active or simply "single" if is not the case.

5.3.3.7 Other global commands

Commands

`addTimeout msec procedure [arg]`

Schedules a Tcl procedure for being called after *msec* milliseconds. If *arg* is specified, it will be passed to the procedure. The specified procedure will be called only once. If necessary, you can schedule it again in the time-out procedure. Example: `addTimeout 10000 echo {10 seconds are over.}`

`all [-selected | -visible | -hidden] [type]`

Returns a list of all **amira** objects currently in the Pool. If *type* is specified, only objects with that C++ class type (or derived objects) are returned. Search can be limited to selected, visible, or hidden objects, respectively. Example: `all -hidden HxColormap`.

`aminfo [-a outfile|-b outfile] AmiraMesh-File`

If used with only a file name as argument, this command will open the file which has to be in

AmiraMesh format and print header information. If used with the -a or -b option, the outputfile specified as argument *outfile* will be written in ASCII (-a) or binary (-b) format, respectively. Thus, *aminfo* can be used to convert binary AmiraMesh into ASCII and vice versa.

`clear`

Clears console window.

`create class name [instance name]`

Creates an instance of an *amira* object like a module or data object. Returns the instance name. Note that data objects are normally not created this way but by loading them from a file. Example: `create HxOrthoSlice MySlice`.

`dso options`

Controls loading of dynamic libraries. The following options are provided:

- `addPath path ...` Adds a path to the list of directories to be searched when loading a dynamic library.
- `verbose {0|1}` Switches on and off debug information related to dynamic shared object loading.
- `open <package>` Tries to load the specified dynamic library. It is enough to specify the package name, e.g., `hxfield`. This name will be automatically converted into the platform dependent name, e.g., `libhxfield.so` on Linux or `hxfield.dll` on Windows.

`echo args`

Prints its arguments to the *amira* console. Use this rather than the native Tcl command `puts` which prints to stdout.

`help arguments`

Without arguments this opens the *amira* help browser.

`httpd [port]`

Start a built-in httpd server. The http server will deliver any document requested. If a requested document ends with `.hx`, *amira* will instead of delivering it execute the file as a Tcl script. This can be used to control *amira* from a web browser. WARNING: This command can create security holes. Do not use this unless behind a firewall and if you know what you are doing.

`limit {datasize | stacksize | coredumpsize} size`

Change process limits. Available on Unix platforms only. Use “unlimited” as size for no limit. The size has to be specified in bytes. Alternatively you can use for example 1000k for 1000 kilobytes or 1m for one megabyte.

`load [fileformat] files`

Load data from one or more files. Optionally a file format can be specified to override *amira*’s automatic file format recognition. The file format is specified by the same label which is displayed in the file format combo box in the *amira* file dialog.

`mem`

Prints out some memory statistics (available on Windows and Irix).

`quit`

Immediately quits `amira`.

`remove {objectname — -all — -All — -selected}`

- `objectname` removes the specified `amira` object.
- `-all` remove all visible objects.
- `-All` remove all objects.
- `-selected` remove selected objects.

`removeTimeout procedure [arg]`

Unscheduled a Tcl procedure previously scheduled with `addTimeout`.

`rename objectname newname`

Changes instance name of an object. Identical to `objectname setLabel newname`, except that it returns 1 if successful, and nothing if unsuccessful.

`sleep sec`

Wait for `sec` seconds. `amira` will not process events in that time.

`source filename`

Loads and executes Tcl commands from the specified file. If the script file contains the extension `.hx` the `load` command may be used as well.

`system command`

Execute an external program. Do not use this unless you know what you are doing.

5.4 `amira` Script File

It is worth noticing that an `amira` network is simply a Tcl script that will regenerate the current `amira` state. Therefore it is often efficient to interactively create an `amira` network, save it with “Save Network”, and use this as a starting point for scripting.

The simplest way to execute Tcl commands in `amira` is to type them into the `amira` console window. This, however, is not practical for multi-line constructs, like loops or procedures. In this case, it is recommended to write the Tcl code into a file and execute the file with the command `source filename`. You can also use the `source` command inside a file in order to include the contents of a file into another file.

Alternatively one can also use the command `load filename` or the *Load* menu entry from the *File* menu and the file browser. Then, however, in order to let **amira** recognize the file format, either the file name must end with `.hx`, or the file contents must start with the header line

```
# Amira-Script-File
```

There are some Tcl files that are loaded automatically when **amira** starts. At startup, the program looks for a file called `.Amira` in the current directory or in the home directory (see Section 4.4 (Start-up script) for details). If no such user-defined start-up script is found, the default initialization script `Amira.init` is loaded from the directory `AMIRA_LOCAL/share/resources` or `AMIRA_ROOT/share/resources`. This script then reads in all files ending with `.rc` from the `share/resources` subdirectory. The `.rc` files are needed to register modules and data types. Therefore one can customize the startup behavior of **amira** by simply adding a new `.rc` file to that directory or by modifying the `Amira.init` file.

Another way of executing Tcl code is to define procedures that are associated with function keys. If predefined procedures with the names `onKeyF2`, `onKeyF3`, ..., `onKeyShiftF2`, ..., `onKeyCtrlF2`, ..., `onKeyCtrlShiftF2`, ... exist, these procedures will be automatically called when the respective key is pressed in the **amira** main window, console window, or viewer window. Note that F1 is reserved for help. To define these procedures, write them into a file and source it or write them into `Amira.init` or in one of the `.rc` files. An example is

```
proc onKeyF2 { } {
    echo "Key F2 was hit"
    viewer 0 viewAll
}
```

Finally, Tcl scripts can also be represented in the GUI and be combined with a user interface. In **amira** this is called a *script object*. There is *separate documentation* for that.

5.5 Configuring Popup Menus

In **amira** all of the modules that can be attached to a data object are listed in the object's popup menu which is activated by clicking on the object's icon with the right mouse button. For some applications it makes sense to customize new modules using Tcl commands after they have been created. Sometimes it also makes sense to add new entries to an object's popup menu, causing a particular script to be executed. This section describes how to achieve these goals by modifying **amira** resource files or creating new ones.

amira resource files are located in the directory `$AMIRA_ROOT/share/resources`, where `$AMIRA_ROOT` denotes the directory where **amira** has been installed. Resource files are just ordinary script files, although they are identified by the suffix `.rc`. When **amira** is started all resource

files in the resources directory are read. In a resource file, modules, editors, and IO routines are registered using special Tcl commands. Registering a module means to specify its name as it should appear in the popup menu, the type of objects it can be attached to, the name of the shared library or DLL the module is defined in, and so on. For example, the *LabelVoxel* module is registered by the following command in the file `hxlattice.rc`:

```
module -name "LabelVoxel" \  
    -primary "HxUniformScalarField3 HxStackedScalarField3" \  
    -check { ![$PRIMARY hasInterface HxLabelLattice3] } \  
    -category "Labelling Compute" \  
    -class "HxLabelVoxel" \  
    -package "hxlattice"
```

The different options of this command have the following meaning:

- The option `-name` specifies the name or label of the module as it will be printed in the popup menu.
- The option `-primary` says that this module can be attached to data objects of type `HxUniformScalarField3` or `HxStackedScalarField3`. This means that *LabelVoxel* will be included in the popup menu of such objects only.
- With `-check` an additional Tcl expression is specified which is evaluated at run-time just before the menu is popped up. If the expression fails, the module is removed from the menu. In the case of the *LabelVoxel* module, it is checked if the input object provides a `HxLabelLattice3` interface, i.e., if the input itself is a label field. Although a label field can be regarded as a 3D image, it makes no sense to perform a threshold segmentation on it. Therefore *LabelVoxel* is only provided for raw 3D images, but not for label fields.
- The option `-category` says that *LabelVoxel* should appear in the *Compute* and *Labelling* submenus of the main popup menu. If a module should appear not in a submenu but in the popup menu itself, the category *Main* must be used.
- The option `-class` specifies the internal class name of the module. The internal class name of an object can be retrieved using the command `getTypeId`. It is this class name which has to be used for the `-primary` option described above, not the object's label defined by `-name`.
- Finally, the option `-package` specifies in which package (shared library or DLL) the module is defined in.

Besides these standard options, additional Tcl commands to be executed after the module has been created can be specified using the additional option `-proc`. For example, imagine you are working in a medical project where you have to identify stereotactic markers in CT images of the head. Then it might be a good idea to add a customized version of the *LabelVoxel* module to the popup menu, which already defines appropriate material names and thresholds. This could be done by adding the following command either in a new resource file in `$AMIRA_ROOT/share/resources` or directly in `hxlattice.rc`:

```

module -name "Stereotaxy" \
    -primary "HxUniformScalarField3 HxStackedScalarField3" \
    -check { ![$PRIMARY hasInterface HxLabelLattice3] } \
    -category "Labelling" \
    -class "HxLabelVoxel" \
    -package "hxlattice" \
    -proc { $this regions setValue "Exterior Bone Markers";
        $this fire;
        $this boundary01 setValue 150;
        $this boundary12 setvalue 300 }

```

The variable `$this` used in the Tcl code above refers to the newly created module, i.e., to the *LabelVoxel* module. Note that the commands are executed *before* the module is connected to the source object for which the popup menu was invoked. Some modules do some special initialization when they are connected to a new input object. These initializations may overwrite values set using Tcl commands defined by a custom `-proc` option. In such a case you can explicitly connect the module to the input object via the command sequence

```

$this data connect $PRIMARY;
$this fire;

```

Here the Tcl variable `$PRIMARY` refers to the input object. The same variable is also used in Tcl expressions defined by a `-check` option, as described above.

Besides creating custom popup menu entries based on existing modules, it is also possible to define completely new entries which do nothing but execute Tcl commands. For example, we could add a new submenu *Edit* to the popup menu of every *amira* object and put in the *Hide*, *Remove*, and *Duplicate* commands here which are normally contained in the *Edit* menu of the *amira* main window. This can be achieved in the following way:

```

module -name "Remove" \
    -primary "HxObject" \
    -proc { remove $source } \
    -category "Edit"

module -name "Hide" \
    -primary "HxObject" \
    -proc { $source hideIcon } \
    -category "Edit"

module -name "Duplicate" \
    -primary "HxData" \
    -proc { $source duplicate } \

```

```
-category "Edit"
```

Of course, it is also possible to execute an ordinary `amira` script or even an `amira` script object with a `-proc` command.

5.6 Registering pick callbacks

A pick callback is a Tcl procedure attached to a module or a viewer. When a pick event occurs on this target, the callback is invoked. Such a callback can be registered by using the Tcl command `setPickCallback` on modules and viewers:

```
<module> setPickCallback <proc> [ <EventType> ]  
viewer <n> setPickCallback <proc> [ <EventType> ]
```

Only one callback can be attached to a given module or viewer. In order to detach the callback, just call the register command with no parameter:

```
<module> setPickCallback  
viewer <n> setPickCallback
```

The optional argument `<EventType>` refers to the kind of event that will invoke the callback. Other events will be ignored. This argument can take the following values:

- `MouseButtonPress`, `MouseButtonRelease` (any mouse button),
- `VRButtonPress`, `VRButtonRelease` (any 3D button),
- `MouseButton1Press`, `MouseButton1Release`, etc. (a specific mouse button),
- `VRButton0Press`, `VRButton0Release`, etc. (a specific 3D button).

The default value is `MouseButton1Press`.

The actual callback procedure `<proc>` is expected to take one argument, which is to be interpreted as an associative array and which encodes all the picking information. The following elements are defined in the argument array:

- *object*, the name of `amira` object the picked geometry belongs to,
- *x*, the x coordinate of picked point,
- *y*, the y coordinate of picked point,
- *z*, the z coordinate of picked point,
- *idx*, the index of picked element,
- *stateBefore*, the modifier state just before event occurs,

- *stateAfter*, the modifier state just after event occurs.

The procedure should return 0 if the picking event was not handled, in which case other callback procedures could be invoked. Here is an example:

```
proc pickCallback arg {
    array set a $arg
    echo "$a(object) : picked element $a(idx)"
    return 1
}
```

Note that any module is free to add specific information to this argument array. All elements can be displayed with:

```
proc pickCallback arg {
    echo "arg = { $arg }"
    return 1
}
```

Thus, some **amira** modules append additional data:

- *VertexView*: *idx* is the picked point index.
- *ClusterView*: *idx* is the picked point index.
- *LineSetView*: *idx* is the picked line index, *pt0* and *pt1* the two points of the picked segment.
- *SurfaceView*: *idx* is the picked triangle index.
- *GridVolume*: *idx* is the picked triangle index, *tetra0* and *tetra1* the adjacent tetrahedra.
- *GridBoundary*: *idx* is the picked triangle index, *originalIdx* the index in the grid, *tetra0* and *tetra1* the adjacent tetrahedra.

Part II

Molecular Pack User's Guide

Chapter 6

Molecular Pack Introduction

Molecular Pack is an amira extension providing support for the visualization as well as the analysis of molecules and dynamic molecular data.

The Molecular Pack documentation is organized into the following sections:

- *Getting started with molecular visualization*
- *Structure and interdependence of the molecular data structures*
- *Essentials for displaying a molecule*
- *Alignment facilities in Molecular Pack*
- *Visualizing dynamic data*
- *Atom expressions*

6.1 First Steps with Molecular Visualization in amira

This chapter gives you an overview of the visualization of molecular data sets. amira is not just able to display a 3D image of the molecule but also provides tools to investigate its distinct parts and properties. After reading the "getting started" introduction you may continue with any of the following tutorials.

- *Getting Started* - first steps
- *Selection, Labeling, and Masking* - exploring a molecule
- *Alignment of Molecules* - visualizing dynamical data
- *Molecular surfaces* - wrapping a molecule
- *Sequential and Structural Alignment* - comparing molecules
- *Molecule Editor* - interactive manipulation of the molecule

- *Molecular Interface* - computing intersection surfaces
- *Measurement* - measuring distances and angles within a molecule

Note: If you want to visualize your own data, please refer to the section about data import in the *amira* User's Guide. That section contains some general hints on how to import data sets into *amira*.

6.1.1 Getting Started with Molecular Visualization

In this section you will learn how to

1. load a molecular demo data set into *amira*,
2. view the molecule with the *MoleculeView* display module,
3. try out various color schemes,
4. select atoms in the viewer using the draw tool,
5. overwrite color scheme colors for selected atoms.

6.1.1.1 Loading Data into the System

In the following introduction we will consider a simple example to explain the basic functions of the *MoleculeView* module. Our example will be a small PDB (Protein Data Bank) structure consisting of 932 atoms in 213 residues.

- Load the file `2RNT.pdb` into the Pool from the directory `data/molecules/pdb`.

A green icon labeled `2RNT.pdb` will appear in the Pool. The green icon represents an object of type *Molecule*. Click on the green data icon with the left mouse button. In the Properties Area below the Pool information about the molecule, such as the number of atoms etc., will be shown. In addition, other ports named *Transformation*, *Selection Browser*, and *Transform*, can be seen; they will be explained in later tutorials.

6.1.1.2 Displaying the Molecule with the MoleculeView Module

The *MoleculeView* module is the basic display module for visualizing molecules. It allows you to display atoms as plates or balls and bonds as lines or cylinders.

- Click again on the green data icon in the Pool, this time using the right mouse button.

A menu, containing several entries and submenus, will appear.

- Select the entry *MoleculeView*.

A new yellow icon labeled *MoleculeView* appears in the Pool. Yellow icons, in general, represent display modules, i.e., modules that visualize objects in the viewer. The blue line between the icons indicates a connection between the objects. In this case the *MoleculeView* module reads data from the object represented by the green icon and visualizes it.

The molecule is now displayed in the viewer as a wireframe. Using the left mouse button you can rotate the object; using the left and middle mouse buttons simultaneously you can zoom in and out. For these mouse operations to work, the viewer must be in *viewing mode*, in which case the mouse cursor is displayed as a hand.

Whenever the *MoleculeView* module is active, the little square on the yellow *MoleculeView* icon is orange. You can deactivate the module by clicking on the square with the left mouse button.

We will explore some basic ports of the *MoleculeView* module.

Mode port: Choose another mode to see both atoms and bonds of the molecule, or just atoms. If atoms are shown, use the *Atom Radius* port to adjust the size of the atoms as desired.

Quality port: If you choose the option *correct*, you can display a correct, i.e. three-dimensional, image of the balls and sticks. Consider the trade-off between correct representation and slower rendering performance. Use the *fast* mode if you want to display a large molecule. If your graphics hardware is fast enough, you can use the *correct* mode even for large molecules.

Complexity port: In order to allow interactive rendering, the default complexity of the scene is rather low. In most cases this will be sufficient. However, if you want to make screen shots or if you have small molecules, you might want to increase the complexity. The *Complexity* port is displayed only if *correct* quality is selected.

6.1.1.3 Changing the Color Scheme

The *MoleculeView* module allows the user to color the molecule according to levels, for example, atom level, residue level, secondary structure level, chain level, or user-defined levels. Each level has a number of attributes, the simplest of which is the *index* attribute. The default color scheme is to color the molecule according to the atom level's attribute *atomic_number*, i.e., the atom's type.

- Click the *Legend* button of the *Color* port to display a small window decoding the colors you see in the viewer window.
- Choose *residues* from the first pop-up menu and *type* from the second menu to color the molecule according to the residue type, here the amino acid type.

The default colormap contains a constant color. Thus, currently all residues of the molecule have the same color.

- To change the colormap, right-click on the color bar of the *Discrete CM* port which has appeared below the *Color* port and select any other colormap.
- Try out different colormaps to find a map that suits your purposes.

Some people prefer the *CPK* color scheme for atoms and the amino acid colors as they are used in the *RasMol* program.

- You can set your preferences in the **amiraEdit** menu by selecting the entry *Preferences*.
- Press the *Molecules* tab and set your preferences. Pressing the *OK* button saves your preferences permanently.

Currently only amino acid colors are predefined. Thus, if the molecule contains residues other than the standard amino acids, those residues will be colored with the default color (black).

6.1.1.4 Using the Draw Tool

The draw tool appears in **amira** in several modules. It enables you to select objects or parts of an object by drawing a line in the viewer.

- Press the *Draw* button of the *Highlighting* port and draw a line in the viewer window around the group of atoms you want to select.

The atoms that were enclosed by the line will be highlighted. If the viewer is the active window, pressing the *d* key is equivalent to pressing the *Draw* button of the *Highlighting* port.

- To unhighlight all atoms, press the *Clear* button of the *Highlighting* port.
- Also try using the keys *Ctrl* and *Shift* while drawing a line.

6.1.1.5 Setting Colors for Selected Parts

The *Define Colors* port allows you to overwrite colors of the color scheme.

- Select some atoms using the draw tool.
- Press the *Set* button of the *Define Color* port.
- Change the current color of the *Color Editor* and press *OK*.

The previously highlighted atoms should now have the color selected in the color editor. This setting will be preserved even if the color scheme is changed. Unfortunately, the new setting will currently not appear in the color legend.

6.1.2 Selection, Labeling, and Masking

You already know how to load files and how to display molecules with the *MoleculeView* module. This tutorial will focus on exploring the structure of a loaded molecule. The tutorial consists of three subsections, in which you will:

1. Explore the possibilities for selecting within a molecule using the *MoleculeView* module.
2. Learn how to use the *MoleculeLabel* module.
3. Get to know the *Molecule Selection Browser*.

For this tutorial we will use the molecule `1IGM.pdb`.

- Load the file `1IGM.pdb` into the Pool from the directory `data/molecules/pdb`.
- Attach a *MoleculeView* module to the green object that has appeared,
- and change the *Mode* port to *balls and sticks*.

6.1.2.1 Interactive Selection with the MoleculeView Module

In the first tutorial you learned how to use the draw tool to select atoms within a molecule. The simplest way to select atoms, however, is to click on the atom of interest. In order to do so, the viewer must be in *interaction* mode. If the mouse cursor in the viewing window is depicted as a hand, you are in viewing mode. To change to interaction mode, you can either

- click on the arrow button in the upper right corner of the viewing window or press the `Esc` button while the viewing window is active.

The mouse cursor will change to an arrow.

- Now click one of the atoms.

The atom you selected should now be highlighted by a red frame around it. If you select another atom, the first atom will be unhighlighted. In order to select more than one atom

- press the `Ctrl` key and keep it pressed while clicking additional atoms.

`Ctrl`-clicking a highlighted atom a second time unhighlights it.

- Now change the *Mode* port of the *MoleculeView* back to *sticks*.
- Select *residues* from the first menu of the *MoleculeView's* *Color* port.
- Choose a suitable colormap from the list of pre-loaded colormaps as described in the first tutorial.

The residues should now be colored according to their type.

- Now select an atom.

As result the whole residue should now be highlighted. The reason for this is that picking is bound to the coloring, by default. For example, if the color scheme is *atoms*, a click on an atom will only

influence the selection for this atom; if the scheme is *residues*, the atom's residue will be selected; if it is *chains*, the atom's chain will be selected, and so on. However, since this is very restrictive you can easily change the selection mode to the most common levels, i.e., atoms, residues, secondary structures, and chains. In order to choose a certain level for the selection you need to press certain keys in advance. Ctrl-Alt-Shift-a chooses the *atoms* level, a click on an atom will only influence the selection for this atom. Ctrl-Alt-Shift-r, Ctrl-Alt-Shift-c and Ctrl-Alt-Shift-s choose *residues*, *chains* and *secondary_structure* level respectively. To switch back to the default behavior, where coloring determines selection, press Ctrl-Alt-Shift-d.

If you select some group and afterwards select a second one from the same level holding down the Shift-key all groups between the two will also be selected. Holding the Ctrl-key pressed while selecting groups allows you to select multiple groups and also to toggle a group when selecting it a second time.

If you do any selection by clicking in the viewer there will be some output in the console window informing you about what you have selected, the amount of output can be customized via the Preferences dialog:

- You can set your preferences in the **amiraEdit** menu by selecting the entry *Preferences*.
- Press the *Molecules* tab and take a look on the options in chapter *Selection Info*:
- **Molecule name** determines that the name of the molecule, to which a selected group belongs, will be printed.
- **Group name** determines that the name of the selected group will be printed.
- **Group attributes** determines that not only the selected group's name but also all attribute values of the group will be printed in the console window.
- **Explicit attributes** restricts the output of attribute values to those attributes that are explicitly named in the text field.
- Pressing the *OK* button saves your preferences permanently.

6.1.2.2 Using the MoleculeLabel Module

So far you have seen that you can select atoms and groups of atoms by clicking on the molecule. The result of the picking is always printed in the console window. This might suffice in some cases. In other cases, however, it is necessary to label the molecule in the viewer so that you can easily track certain groups you are interested in. If you want to use this tool, here is how you do it.

- Click again on the icon *IIGM.pdb* with the right mouse button.
- Select *MoleculeLabel* from the *Display* submenu.

A second yellow icon labeled *MoleculeLabel* should have appeared in the Pool. By default, all clicks will now be handled by the *MoleculeLabel* module. This means that instead of highlighting groups when clicking on them they will now get labeled.

- Type `Ctrl-Alt-Shift-r` while the viewing window is active.
- Click on the molecule in the 3D viewer.

This action should cause the residue of the selected atom to get labeled. Pressing `Ctrl-Alt-Shift-a` and clicking the same atom will result in the selected atom being labeled. The `Ctrl-` and `Shift-` keys have the same effect as when selecting groups.

- Label a few residues and atoms.
- Click on the *MoleculeLabel* icon to view its ports in the Properties Area.
- Change the color of the atom labels by pressing the color button of the *Color* port and selecting a different color in the color editor.
- Now select *residues* in the *Levels* port. The *Attributes*, *Level Option*, *Buttons*, *Font Size*, and *Color* ports now affect the residue labels.
- Increase the font size (which only increases the font size of the current level, i.e., residues).
- Select the *name* entry in the second menu of the *Attributes* port.

The selected residues will now be labeled by their types and names. In order to understand the *Options* port of the *MoleculeLabel* module

- deselect the last option, *replace attributes*,
- and set the entry of the second *Attributes* menu back to *disabled*.
- Now select a new residue, holding the `Ctrl-` key down.

The new residue will only be labeled by its type. In contrast, the old residues are still labeled with type and name.

If the first option of the last port, *handle clicks*, is deselected, mouse clicks will be handled just as if the module *MoleculeLabel* did not exist.

- Deselect the *handle clicks* option.
- Select any residue.

The residue is selected, not labeled.

6.1.2.3 Molecule Selection Browser

In this section you will learn the basics about the *molecule selection browser*, which is a very important feature for the investigation of a molecule. The selection browser is a flexible tool for selecting those parts of the molecule that you are interested in. Moreover, it allows you to easily combine different viewing modules into one viewer.

- Select the green icon *IIGM.pdb* by clicking on it.

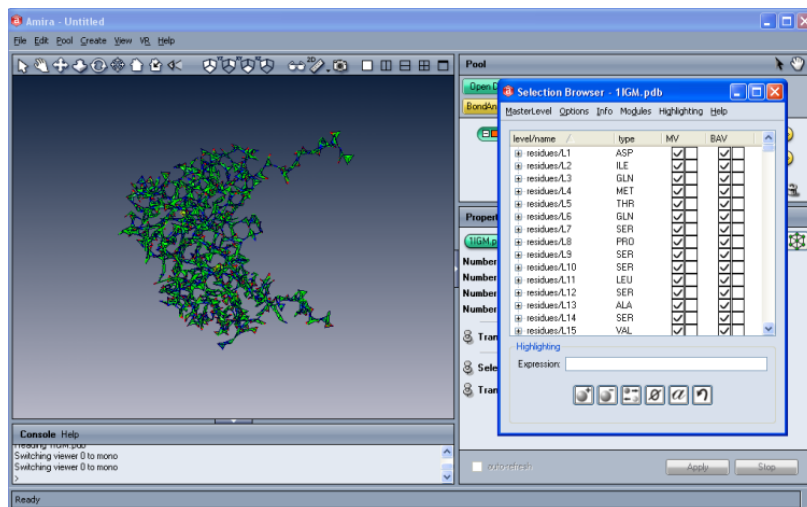


Figure 6.1: On the left, MoleculeView and BondAngleView displaying the molecule simultaneously. On the right, the Selection browser after connecting two viewing modules to the molecule.

- Press the *Show* button of the *Selection Browser* port to open the browser.

The scroll view of the browser currently contains three columns, one with the heading *level/name*, the second with the heading *type*, and the third with the heading *MV* which stands for *MoleculeView*. In the first column, all residues of the molecule are displayed. The second column shows the residue type.

- Connect a *BondAngleView* to *1IGM.pdb* by right-clicking the icon and choosing *BondAngleView* from the *Display* submenu.

You will observe a new column in the selection browser, representing the *BondAngleView (BAV)* (see Figure 6.1).

Changing the Appearance of MoleculeView and BondAngleView

In order to get an image similar to Figure 6.2, we first need to set the ports in the *MoleculeView* and the *BondAngleView* modules correctly. We begin with the *MoleculeView* module.

- Set the *Mode* port to *balls*.
- Set the *Quality* port to *correct*.
- Set the *Atom Radius* port to 1.0.

For the *BondAngleView* continue as follows:

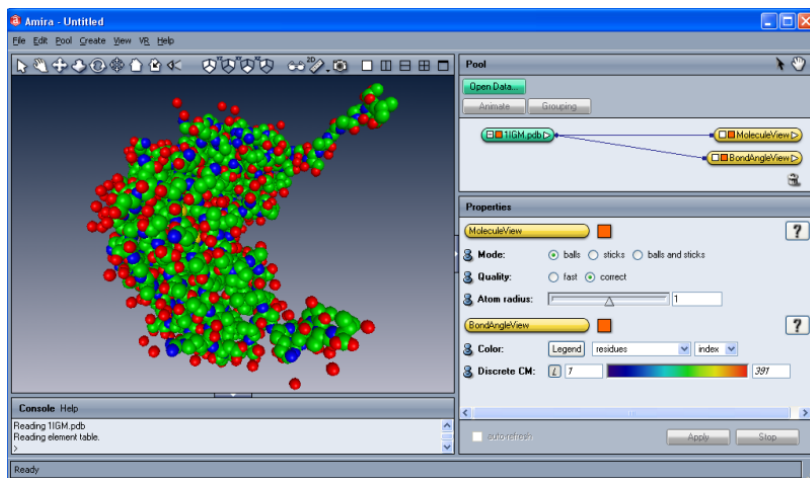


Figure 6.2: Setting the ports in the viewing modules.

- Select the *residues* entry from the first menu of the *Color* port.
- From the second menu of *Color* port, select the *index* entry.
- Right-click on the color bar of the *Discrete CM* port and select the colormap *physics.icol*, if this colormap is pre-loaded. Otherwise load it from the directory *data/colormaps*.

We now only see the *MoleculeView*, since the triangles of the *BondAngleView* are hidden by the van der Waals spheres. In order to combine the two viewing modules into one image, continue with the next section.

Highlighting and Masking with the Selection Browser

We now want to use the selection browser to display parts of the molecule with the *MoleculeView* module and the rest with the *BondAngleView* module. In this section we will only use the selection browser, so all menu names etc. refer to this window.

- Select *chains* as new master level from the *MasterLevel* menu.

You will now see three entries in the first column: *chains/L*, *chains/H*, and *chains/W*. This means that the level *chains* contains three groups, named *L*, *H*, and *W*. In order to view the group *chains/L* with the *MoleculeView* and the other two groups with the *BondAngleView*,

- remove the check marks for *chains/H* and *chains/W* from the column with heading *MV* by clicking on them.

To deselect the group *chains/L* for the *BondAngleView*,

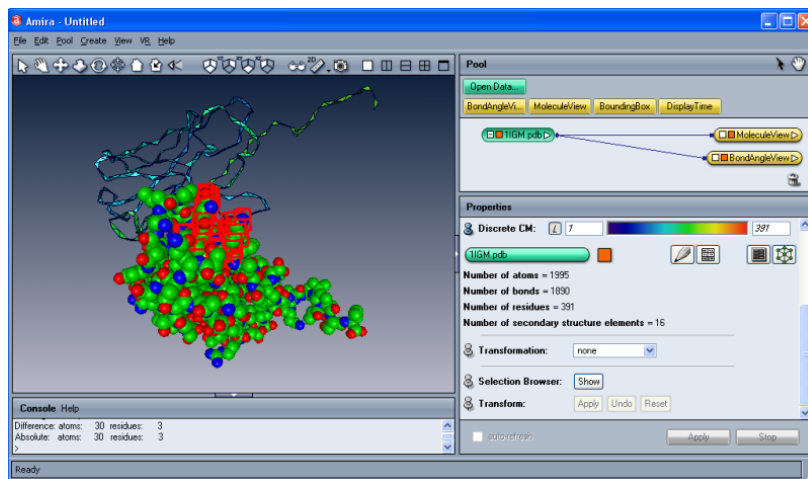


Figure 6.3: Displaying parts of the molecule with the *MoleculeView* module and the rest with the *BondAngleView* module.

- click on the respective box in the column *BAV*.

Now you should see an image that is pretty close to that in Figure 6.3. However, the *BondAngleView* displays more triangles than in the figure. The *BondAngleView* in Figure 6.3 only displays *backbone* atoms. In order to achieve this

- right-click on the heading labeled *BAV*.

A pop-up menu as in Figure 6.3 should appear.

- Select *Backbone* from the *Restrict to* submenu.

The side-chain atoms should not be visible anymore. Next, you should explore the group *chains/L* a bit further.

- Click on the little icon left to it.

After this action the residues contained in chain *L* will have appeared.

- Click on the little icon left to the residue *L1*.
- Type `atoms/L4-L33` in the text field labeled *Expression*, below the scroll window, and press the *Replace* button.

Atoms *L4* to *L8* and residues *L2*, *L3*, and *L4* are highlighted in red. The residue *L1* and the chain *L* are highlighted in green. The color *red* means that the whole group is selected, i.e., all its elements. *Green*

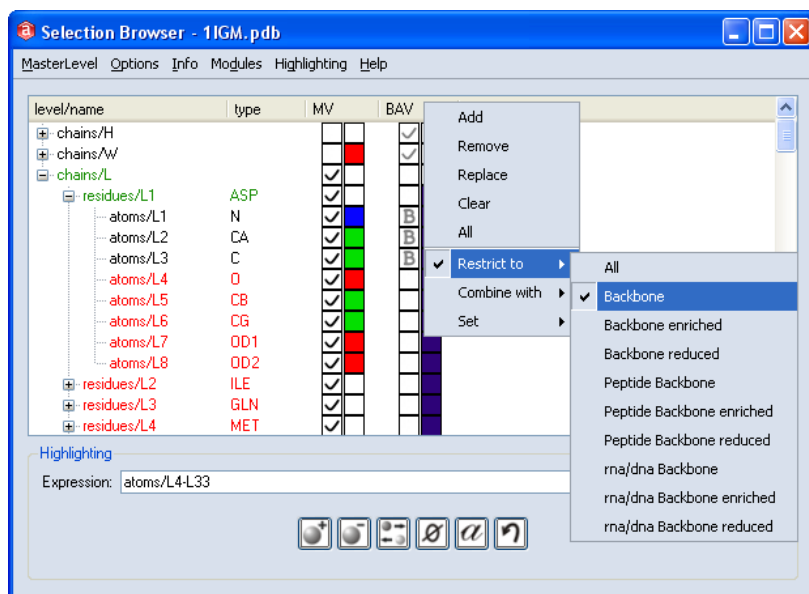


Figure 6.4: Molecular browser dialog.

denotes that the group is partially selected. See Figure 6.4.

To get familiar with the browser, play around with it. If you need further information, just press the F1 button in the browser window or see the description of the *Selection Browser*.

6.1.3 Alignment of Molecules

In this section you will learn how to

1. align molecules by considering selected atoms,
2. compute a *mean molecule*,
3. compute a *configurational density*,
4. compare metastable molecular conformations.

6.1.3.1 Comparing two Molecules

In this section we will compare two different three-dimensional structures of the same molecule.

- Load the data file `alkane.zmf` from the directory `data/molecules/alkane`.
- Select *MolTrajectory* from the pop-up menu of the *alkane.zmf* icon.

- Select *Molecule* from the pop-up menu of the *MolTrajectory* icon.
- Attach a *MoleculeView* module to the molecule.
- Repeat the last two steps, creating two new objects, *Molecule2* and *MoleculeView2*

The *alkane.zmf* icon represents a bundle of molecular trajectories, in this case butane, pentane, and hexane. By attaching a *MolTrajectory* to it, we extract a single trajectory. By default, the first trajectory, which is butane, is selected. We can now extract single time steps from the trajectory by attaching a *Molecule* object to the trajectory. We have done this twice since we want to compare two time steps with each other. Currently, both molecules extract the same time step. This is the reason for only seeing a single molecule. In order to get more information about the data structures, go to the section on *molecular data structures*.

- Select the *MoleculeView* icon and change the *Mode* port to *balls and sticks* and the *Quality* port to correct.
- Repeat this action for the *MoleculeView2*.
- Select the *Molecule2* icon and change the value of the *Time* port to 2.

You should now see two butane molecules in the viewer, slightly displaced.

- Right-click on the white square at the far left side of the *Molecule2* icon and select *AlignMaster* from the pop-up menu.
- Connect the blue line to the *Molecule* icon.
- Select the *Molecule2* icon.
- Press the *all* button of *Molecule2's* *Select* port.

You have now aligned the molecules *Molecule* and *Molecule2* using a least squares fitting of all atoms. This only works for molecules with the same number of atoms. In the following we will align the two molecules by considering only three carbon atoms.

- Deactivate the *MoleculeView* by clicking on the orange square of the *MoleculeView* icon.
- Ctrl-click on three consecutive carbon atoms.

The frame of a red cube should appear around each of them.

- Activate the *MoleculeView* by clicking on the gray square on the *MoleculeView* icon.
- Select the *Molecule2* icon and press the *in slave* button of the *Select* port.

The three selected atoms should now fit better onto the corresponding atoms of the object labeled *Molecule*.

6.1.3.2 Computing a Mean Molecule

In this section we will compute a mean molecule of a metastable molecular conformation of butane.

- Remove the objects *Molecule* and *Molecule2*.
- Select the *MolTrajectory* icon.
- Load the data file *but_cluster_3_1.idx* from the directory *data/molecules/alkane*.

but_cluster_3_1.idx is a subtrajectory of *MolTrajectory*, i.e., a subset of all configurations in the trajectory.

- Attach a molecule to the icon *but_cluster_3_1.idx*.
- Select the *MeanMolecule* entry from the *Compute* submenu of *but_cluster_3_1.idx*'s pop-up menu.
- Right-click with the mouse on the white square at the far left side of the *MeanMolecule* icon and select *AlignMaster*.
- Connect the *AlignMaster* port to the *Molecule* icon by attaching the blue line to it.
- Select the *MeanMolecule* icon and press the *all* button of the *Select* port of the *MeanMolecule* module.
- Press the *Apply* button to compute the mean molecule.
- Visualize the mean molecule *but_cluster_3_1.mean* by attaching a *MoleculeView* to it.
- Connect the *AlignMaster* port of the module *MeanMolecule* to the *but_cluster_3_1.mean* icon.
- Press the *Apply* button of the *MeanMolecule* module two or three times.

You have now computed a mean molecule of the subtrajectory *but_cluster_3_1.idx*. The *AlignMaster* is used to align all time steps before accumulating the atom positions. Changing the *AlignMaster* to the previously computed mean molecule and repeating the computation of the mean molecule will improve it. This procedure should converge.

A better way to compute a mean molecule is to use the module *PrecomputeAlignment*, using the *Multiple Alignment* option of the *Mode* port. This module creates an object containing a transformation for each structure in the trajectory. This object can then get connected to the *PrecomputeAlignment* connection port of the *MeanMolecule* module.

6.1.3.3 Computing and Visualizing a Configuration Density

For the subset of configurations contained in the subtrajectory *but_cluster_3_1.idx*, we will now compute a *configuration density* and visualize it with the *Isosurface* display module.

- Select the entry *ConfigurationDensity* from the *Compute* submenu of *but_cluster_3_1.idx*'s pop-up menu.

- Connect the *AlignMaster* port of the *ConfigurationDensity* icon to the *but_cluster_3_1.mean* icon.
- Select the *ConfigurationDensity* icon.
- Press the *all* button of the *Select* port.
- Press the *Field* button of the *Compute* port to compute the density.
- Visualize the created scalar field *but_cluster_3_1.scalar* with an isosurface by first selecting the *Isosurface* entry from the *Display* submenu of the icon's pop-up menu,
- second, setting the *Isosurface*'s *Threshold* value to 0.1,
- and third, pressing the *Apply* button.
- Try different *Threshold* values.

6.1.3.4 Comparing Metastable Molecular Conformations

The set of configurations in the subtrajectory *but_cluster_3_1.idx* belongs to a metastable conformation of butane. In this section we will compute the density of a second metastable conformation and compare the two with each other.

- Select the *MolTrajectory* icon.
- Load the data file *but_cluster_3_2.idx* from the directory *data/molecules/alkane*.
- Compute the mean molecule of the subtrajectory *but_cluster_3_2.idx* by repeating the steps described above for the subtrajectory *but_cluster_3_1.idx*.
- Visualize the second mean molecule with the *MoleculeView* module.
- Select three consecutive carbon atoms in the second mean molecule as was done in the first tutorial.
- Attach the *AlignMaster* of the object *but_cluster_3_2.mean* to the first mean molecule *but_cluster_3_1.mean*.
- Select the *but_cluster_3_2.mean* icon and press the *in slave* button of the mean molecule's *Select* port.

The two mean molecules should now be aligned to each other, i.e., the three selected carbon atoms should superimpose.

- Compute the density for the second subtrajectory using the *but_cluster_3_2.mean* molecule as *AlignMaster*.
- Visualize the computed density with an *Isosurface* module.
- Double-click on the colormap of the *Isosurface2* module and select a different color to better distinguish the two isosurfaces from each other.

You should now clearly see how the two conformations of butane differ. For larger molecules it might be interesting to color the isosurface according to the atom's colors. This can be done using the

same *ConfigurationDensity* modules by selecting the *Color Field* option of the *Field* port. Attach the computed color field to the *ColorField* connection port of the *Isosurface* module.

Notice that you can also visualize the densities with the *Voltex* module of the *Display* submenu.

6.1.4 Molecular Surfaces

In section 6.1.1 of this tutorial you have seen how to visualize molecules with the *MoleculeView* module. Section 6.1.2, on selection, labeling and masking, showed you how to use *amira*'s facilities to select, mask, and label certain parts of the molecule. In this tutorial we will

1. compute a molecular surface with the *CompMolSurface* module,
2. compute the molecular surface for a restricted set of atoms,
3. compute partial surfaces,
4. explore the *MolSurfaceView* module,
5. get to know the picking facilities of the *MolSurfaceView*.

6.1.4.1 Compute Molecular Surfaces

In this section you will learn how to use the *CompMolSurface* module to compute molecular surfaces with different resolutions. You will also become familiar with some of the module's basic ports.

- Load the data file *2RNT.pdb* from the directory *data/molecules/pdb*.
- Attach the *CompMolSurface* module to the green icon by selecting the corresponding entry from the *Compute* submenu of *2RNT.pdb*'s pop-up menu.
- Select the *CompMolSurface* icon.
- Press the *Apply* button.
- Attach the *MolSurfaceView* module to the newly created green icon, *2RNT-surf*.

You should now see a gray solvent excluded surface which is still pretty coarse. If you want a surface with better resolution,

- increase the number of points per \AA^2 in the *CompMolSurface* module.
- Press the *Apply* button again.
- Try different resolutions.
- Now select the *no duplicate points* option in the *Options* port.
- Press the *Apply* button.

Due to the last action, some sharp edges will have disappeared. If there are no duplicate points at sharp edges, the surface normals of the adjacent triangles will be interpolated, thus leading to a smoothing,

which in this case might be undesired. However, this option is important if it is necessary to have a completely closed surface.

- Change the *Quality* port to *faster*.
- Press the *Apply* button.

You should observe that some atoms disappear. This is due to the fact that now only one surface component will be computed. If the molecular surface consists of only one component, the whole surface will be computed. Since the underlying algorithm does not need to touch every single atom, but approximately only every second atom, the algorithm is much faster. If performance is an issue, you might consider using this option.

6.1.4.2 Compute Partial Surfaces

In order to compute partial surfaces we need to select the atoms for which we want to compute their surface contribution.

- Open the selection browser. If you do not know how to do this, take a look at the *second tutorial*.
- Type `within(residues/HET105, 5)` in the browser's *Expression* command line.
- Press the selection browser's *Replace* button.

All atoms within a distance of 5 Å of the *HET105* residue will now be selected.

- Reset the *CompMolSurface*'s *Quality* port back to *correct*.
- Select *partial surface* from the *Options* port of *CompMolSurface*.
- Press the *Apply* button.
- Select *adjacent patches* from the *Options* port of *CompMolSurface*.
- Press the *Apply* button.

The last action causes the partial surface to be expanded by the toroidal patches adjacent to the partial surface.

6.1.4.3 Molecular Surface of a Restricted Set of Atoms

The molecule *2RNT.pdb* contains some water molecules, which are in most cases not desired when computing the molecular surface. In order to exclude the water molecule from the surface computation

- open the selection browser again.
- In the selection browser scroll to the end of the list of residues.
- Click on one of the strings *HOH* in the *type* column.

All residues of type *HOH* should now be highlighted.

- In the browser, right-click on the heading *CMS* which stands for *CompMolSurface* and select the entry *Remove*.
- Deselect *partial surface* in the *CompMolSurface* module.
- Press the *Apply* button.

For the new surface only those residues that have a check mark in the selection browser were considered. You may combine this procedure with the partial surface computation described above.

6.1.4.4 Exploring the MolSurfaceView

The *MolSurfaceView* module is already attached to the molecular surface. A second connection exists to the molecule *2RNT.pdb*, from which data is read to enhance the visualization.

- Compute the whole molecular surface with a resolution of 2 points per Å².
- Click on the *MolSurfaceView* icon to see its user interface in the Properties Area.
- Select *molecule* for the *Color Mode*.
- Change the *Color* port's first menu entry to *residues*.
- Select an appropriate colormap for the *Discrete CM* port by right-clicking on the color bar.
- Switch to interaction mode by clicking on the arrow button in the upper right corner of the viewing window.
- Click on the surface in the viewing window.

All triangles belonging to the picked triangle's residue will be highlighted. If the triangle belongs to two or even three (maximum) residues, all of those residues will be highlighted.

Clicking on the surface with the middle mouse button displays information about the atom you clicked on in the upper left corner of the viewing window as long as you keep the mouse button pressed. If you **Ctrl**-click, the information will remain displayed even after releasing the mouse button until the next mouse click on the surface.

- Press the *Highlighting* port's *Clear* button to remove the selection.
- Change the *Pick Action* port to *clipping*.
- Pick any triangle of the surface.

All triangles further away from the picked triangle than the distance given by the *Selection Distance* port will be cut off. All triangles within this distance will remain, however, only if they are connected to the picked triangle without leaving the sphere around the picked point.

- Press the *All* button of the *Buffer* port to display the whole surface.
- Change the *Pick Action* port to *surface*.
- **Shift**-click on the surface in the viewing window.

All clicks on the surface will now be handled as you might be familiar with from the *SurfaceView* module.

6.1.5 Sequential and Structural Alignment

In this tutorial you will become familiar with the *AlignSequences* and *AlignMolecules* modules.

The *AlignSequences* tool facilitates the comparison of two sequences. It can be applied to both proteins and nucleic acids except for t-rna molecules containing modified bases. The *AlignSequences* module is *not* highly advanced, but it might suffice for some purposes.

Having done the sequential alignment, you can use the correspondence produced by the sequence alignment to do a structural alignment of the molecules.

- Load the files *1IGM.pdb* and *2JEL.pdb* from the directory *data/molecules/pdb/* and connect a *MoleculeView* to each of them.

You should know how to do this from the *first tutorial*. For both *MoleculeViews*

- select *residues* from the first menu of the *Color* port.

A colormap (*Discrete CM*) with constant color will appear. To distinguish the molecules, choose a different color for one of them. In order to do so

- double-click on the colorbar (the *Color Dialog* should appear) and
- change the current color by dragging and dropping any of the custom colors.
- Press the *OK* button of the color editor.

6.1.5.1 Sequence Alignment

We will now use the *AlignSequences* module to align the sequences of the two molecules. In the next section we will use the associated amino acid pairs for a structural alignment.

- Right-click on the *1IGM.pdb* icon and select *AlignSequences* from the *Compute* submenu.
- Right-click the white square on the far left side of the *AlignSequences* icon.

A pop-up menu displaying all connection ports opens.

- Select *MoleculeB* from it and connect the port to the *2JEL.pdb* icon by clicking on it.

There should now be two blue lines connecting the *AlignSequences* icon with the *1IGM.pdb* and *2JEL.pdb* icons, respectively.

- Select the module *AlignSequences* by clicking on its icon in the Pool.

- Choose *semiglobal* from the second menu of the *Align Type* port.
- Press the *Apply* button.

If the alignment has been successful, a window displaying several slightly different alignments will appear.

- Press the *AcceptAll* button. Then press the *Close* button.

This action results in the alignments being written to the molecules as new levels. In order to check this,

- type `1IGM.pdb list` in the *amira* console window.

In addition to levels such as *atoms*, *bonds*, *residues*, etc. you will also see levels named *semiGlobalSeqAlign1* to *semiGlobalSeqAlign10*. *2JEL.pdb* has the same levels with an equal number of groups.

6.1.5.2 Align Molecules by using the Mean Distance Criteria

We can now use the levels *semiGlobalSeqAlign** for a structural alignment. In order to do so,

- right-click the icon *1IGM.pdb* and select *AlignMolecules* from the *Alignment* submenu.
- Right-click the white square at the far left side of the *AlignMolecules* icon, select the *MoleculeB* entry, and connect the module to *2JEL.pdb*.
- Select the *AlignMolecules* icon to make it appear in the Properties Area.
- Select any of the levels named *semiGlobalSeqAlign** in the *AlignLevel* port and press the *Apply* button.

If you want to follow the alignment process,

- click the *show alignment* option before pressing the *Apply* button.

Compare your result to the online demo.

6.1.6 Editing of molecules

An essential tool for manipulating the molecular data structure from *Molecular Pack* is the *Molecule Editor*. It allows adding new bonds or changing the overall topology of the molecule as well as manipulating Cartesian or internal coordinates.

For each of these tasks, there is an example in the following section to give you an easy “learn by doing” introduction to the available features. Each action performed with the editor affects all currently selected atoms. Thus, it is necessary for you to be familiar with the functions of the *SelectionBrowser* beforehand (see section 6.1.2 on selection, labeling, and masking).

- Load the file `1HVRm.pdb` from the directory `data/molecules/pdb/` and connect a *MoleculeView* module to it.
- Select the *balls and sticks* representation for the *MoleculeView*.

The data structure loaded is an enzyme of HIV in complex with the inhibitor XK263.

6.1.6.1 Invoking the Molecule Editor

To start the editor

- select the `1HVRm.pdb` object in the Pool and press the Molecule Editor button (pencil icon) in its user interface.

The molecule editor interface is now visible. However, for the tasks we want to perform, we also need the selection browser, which is opened by

- pressing the *Show* button of the *Selection Browser* port of *1HVRm.pdb*.

6.1.6.2 Adding Bonds to a Part of a Molecule

The inhibitor XK263 is currently without bonds. To add bonds, carry out the following steps:

- Open the selection browser and select the residue with the type *XK2* (it is at the end of the list).
- Switch to the *Tools* tab in the molecule editor and press the *bond-length table* button in the *Mode* section. Then, press the *add* button in the *Action* section.

Adding bonds in the *bond length table* mode will create new bonds in the selected part of the molecule by comparing the distances between the atoms with a table of average bond lengths. The result should now look like Figure 6.5.

6.1.6.3 Splitting the Molecule

We now want to split the molecule into inhibitor and protein.

- If it is not currently selected, reselect the *XK2* residue in the selection browser.
- Press the *Split* button on the *Tools* tab of the molecule editor.

You will notice that the atoms of the inhibitor are no longer displayed by the *MoleculeView* and that a new object, *1HVRm2.pdb*, has appeared in the Pool. This object contains the previously selected atoms of the ligand.

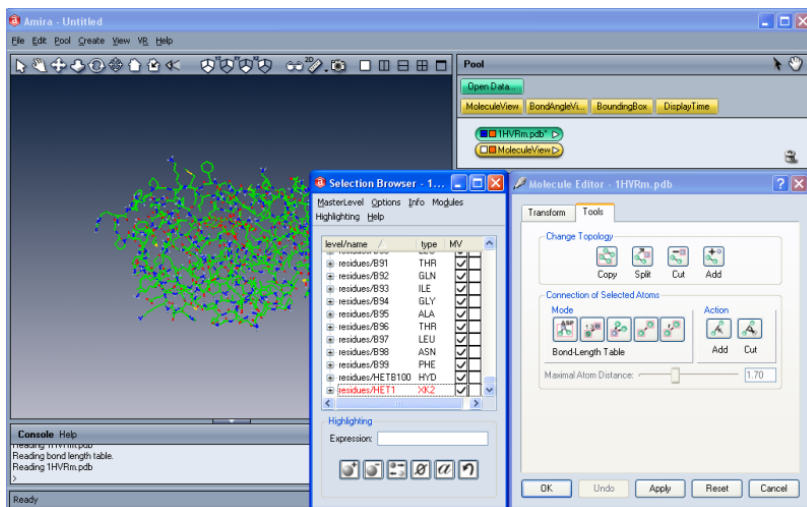


Figure 6.5: Adding bonds to the ligand

6.1.6.4 Adding another molecule

The protein of the 1HVR entry is a protease which uses up one water molecule to split a polypeptide. We now want to add the water to the active site of the enzyme.

- Load the file `h2o.pdb` from the directory `data/molecules/pdb`.
- Go to the molecule editor and press the *Add* button in the *Change Topology* section.
- In the window that opens, all other molecules in the Pool will be shown. Select the `h2o.pdb` molecule and press *OK*.

The atoms of the water molecule have now been copied to the *1HVRm.pdb* object.

6.1.6.5 Moving Parts of the Molecule

To concentrate our view on the region of interest we will reduce the molecular view to amino acids A25 and B25 between which the water molecule should be placed.

- Type `r/name=?25` OR `r/type=H2O` into the selection browser and press the *Add* button.
- Now move your mouse on the *MV* heading, press the right mouse button, and activate the *Replace* option.

With only the residues A25 and B25 and the water molecule displayed, all that is left to do is to drag the water molecule to its correct location.

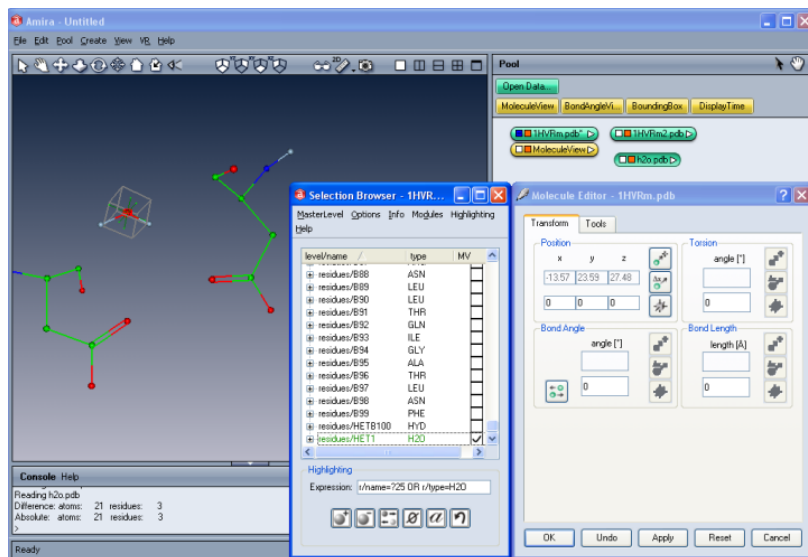



Figure 6.6: Moving the water molecule to the desired location

- Select the water molecule in the selection browser (the residue at the end of the list).
- Switch to the *Transform* tab of the molecule editor.
- Show the position of the transform dragger by pressing the  button in the *Position* section of the *Transform* tab.
- Left-click on the dragger and hold the mouse button down. You can now translate the dragger in different planes depending on the side you clicked on. Move the water molecule between the two amino acids as shown in Figure 6.6.
- To reorient the water molecule, click on the green knobs of the dragger. They allow the dragger to rotate in different planes.

Repeat the steps described above until you are satisfied with the result.

To apply the changes that you made to the edited molecule, you must press the *Apply* button or end the editing by using the *OK* button.

6.1.7 Molecular Interfaces

This tool is particularly interesting for visualizing contact areas between parts of a single molecule or between different molecules, e.g., enzymes and their ligands. In this example we will consider the second case.

- Please load the file `2RNT.pdb` from the directory `data/molecules/pdb`.

6.1.7.1 Defining groups

In this section we will create a new level, called *interface*, for the molecule and define two groups of the level, *substrate* and *receptor*, respectively.

- Select the `2RNT.pdb` icon in the Pool by clicking on it.
- Now click in the console window to activate it, and press the TAB key.

The name of the selected icon, `2RNT.pdb`, should appear in the console window. If it does not,

- please type `2RNT.pdb`.
- On the same line, type `define interface/substrate residues/HET105`

The following text should now be written in the console window

```
2RNT.pdb define interface/substrate residues/HET105
```

- Press the ENTER key.

You have now defined the group *substrate* in the level *interface*. The group consists of the residue HET105. We will now define the group *receptor*.

- Press the TAB key again, then type `defregexp interface/receptor NOT residues/HET105 AND NOT residues/type=HOH`

With `NOT residues/HET105` we exclude the substrate and with `NOT residues/type=HOH` we exclude all water molecules. Thus, the receptor is defined as consisting of all atoms not belonging to either the substrate or any water molecule.

If you now list all levels by typing

- `2RNT.pdb list`

you will see an *interface* level containing two groups.

- Type `2RNT.pdb list interface`

and all groups of the *interface* level will be printed in the console window.

6.1.7.2 Computing the Interface

We will now compute the interface between the receptor and the substrate. The interface between two molecules is defined as the surface equidistant to both molecules. For the approximation we compute,

this might not be exactly true for all points on the surface. The module to compute the interface is called *CompMolInterface*.

Now, to compute the interface,

- right-click the icon *2RNT.pdb* and select *CompMolInterface* from the *Compute* submenu.
- Select the icon labeled *CompMolInterface* in the Pool.
- Choose *interface* from the *Levels* menu of the *CompMolInterface* module.
- Press the *Apply* button.

Two objects result from this action, an interface object of type *MolSurface*, *2RNT-interface.surf*, and a distance field of type *UniformScalarField3*, *2RNT-distance.field*.

6.1.7.3 Visualizing the interface

Finally, we will view the computed interface in the viewing window with the *MolSurfaceView* module.

- Right-click on the *2RNT-interface.surf* icon and select *MolSurfaceView*.
- Right-click on the white square at the far left side of the *MolSurfaceView* icon and connect the *ColorField* port with the distance field *2RNT-distance.field*.

The user interface of the *MolSurfaceView* module should be visible in the Properties Area.

- Choose the *field* option from the *Color Mode* port, resulting in a new port, *Colormap*, appearing in the user interface.
- Right-click on the colormap bar and choose any colormap.
- Play around with the coloring by changing the colormap range. The full range of values can be seen when you select the *2RNT-distance.field*, i.e., click on it.
- Also, change the *Cutoff distance* in the *CompMolInterface* module, e.g., to 1.0, and the *Voxel size*, e.g., to 0.5.
- See what happens when you press the *Apply* button of the *CompMolInterface* module.

Warning: If you choose a very small voxel size, the computation might take very long. Also, there might not be enough memory to store such a large distance field. Usually, 1.0 or 0.5 are good values.

After having done the steps described above, what you see should be similar to the Nuclease demo on the demo pages. The interface here, however, has been generated from two separate files.

6.1.8 Measurement

In this tutorial you will learn how to measure distances and angles between atoms in a molecule. For this tutorial it is necessary for you to have already done the tutorial [6.1.1](#).

- Load any molecule from the directory `data/molecules/` and connect a *MoleculeView* to it.
- Select the *MoleculeView* by clicking on its icon in the Pool.
- Select *balls and sticks Mode* and *fast Quality*.
- Next, right-click on the *MoleculeView* icon, and select *Measurement* from the pop-up menu.

The user interface of the *Measurement* tool should now be visible in the Properties Area. An *Info* port tells you that you need to select 2, 3, or 4 atoms. In order to do so, switch to interaction mode by

- pressing the ESC key or by pressing the arrow button in the upper right corner of the viewing window.

You can now select single atoms in the viewer by clicking on them. If you click on one atom, the previously selected atom will be deselected. By using the Ctrl-key, you can select more than one atom.

- Select 2, 3, or 4 atoms and observe what is being displayed in the user interface of the *Measurement* module.

The Ctrl-key is also used to deselect atoms.

6.2 Molecular Data Structures

Several molecular file formats including, for example, PDB, Tripos, and UniChem, can be read and written by *Molecular Pack*, other file formats, such as CHARMM, can only be read. The information read from the data files will be stored in different types of objects: *Molecule*, *MolTrajectory*, and *MolTrajectoryBundle*.

Molecule. Single molecular configurations are represented as data objects of type *Molecule*. This kind of object can either be created by loading a file containing a single molecular structure or by attaching it to an object of type *MolTrajectory*, in which case it will act as a projector, extracting one of the trajectory's 'time steps', i.e., a single structure.

Trajectory. The *MolTrajectory* data structure represents a series of molecular configurations of the same molecule. Again two cases are possible. Either the trajectory is loaded directly from a file. Or it can be attached to an object of type *MolTrajectoryBundle*, extracting one of the trajectories contained in that bundle.

Bundle of trajectories. A trajectory bundle reflects the case of a file containing more than one molecular trajectory. If such a file is loaded into *amira*, an object of type *MolTrajectoryBundle* will be created. A single trajectory can be accessed by attaching a *MolTrajectory* object.

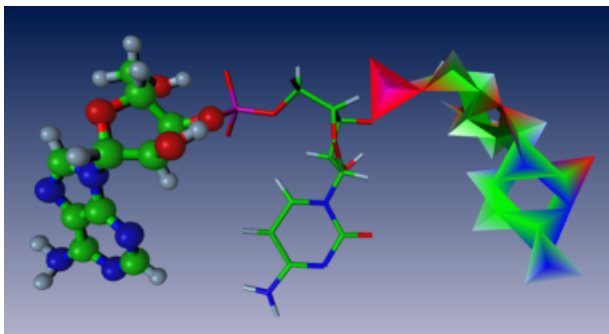


Figure 6.7: MoleculeView (left and middle) and BondAngleView (right) display the molecule simultaneously.

6.2.1 Internal Structure of Molecules

An object of data type *Molecule* contains information about the structure of a molecule and the atomic coordinates of one of its configurations. The mandatory part of structural information concerns the number and types of all atoms contained in the molecule. All the topological information is organized in levels which are cliques of groups. Each group contains other groups or atoms. The simplest example is the level *bonds*, which consists of groups of two atoms.

Some levels can build hierarchies. For example, a residue consists of a number of atoms, and a couple of residues may form a secondary structure. These levels and groups can be defined by file readers or interactively via *Tcl* commands.

6.3 Displaying Molecules

Molecules can be visualized with the *MoleculeView*, *BondAngleView*, *SecStructureView*, or *TubeView* modules. A sample output of the first two modules is shown in Figure 6.7.

In addition, there are two modules for generating molecular surfaces. The *CompMolSurface* module enables you to generate the *solvent accessible*, *solvent excluded*, and *van der Waals surfaces* of a molecule. The *CompMolInterface* module can be used to generate intra- and intermolecular interfaces, such as between single atoms or residues, or between two molecules, respectively.

6.3.1 Coloring Molecules

The atom-oriented modules *MoleculeView*, *BondAngleView*, *ConfigurationDensity*, and *MolSurfaceView* allow a coloring on a per-atom basis, i.e., each atom is assigned a color. Balls in the *MoleculeView* will have the corresponding colors. Sticks will be split into halves colored according to their attached atoms. In the *BondAngleView*, atom colors are assigned to the vertices and color is interpolated over the triangles.

To determine the coloring you can choose a level, e.g., atoms, residues, secondary structures, or chains. Furthermore, you can choose one of the level's attributes. The coloring will then be done according to the group's attributes such that all atoms of the same group will have the same color.

Color



A molecule may be colored according to various schemes. For example, each atom of a specific type may have the same color. Another possibility is to give all atoms belonging to a certain group the same color. The first menu of the *Color* port specifies the group level, e.g., atoms, residues, secondary structures, or chains. The second menu allows you to decide which attribute of the specified level should be considered to color its groups.

If you press the *Legend* button, a separate window will be opened, which displays a legend of the coloring, i.e. a table associating colors with attribute values according to the current coloring. Clicking on the text to the right of a color will select all atoms with this color.

Continuous CM



This colormap is used to map values of float attributes to colors. For optimal mapping, the colormap range must be set correctly. By default, the range is set to the minimum and maximum values that occur using the chosen color scheme, if the button in front of the first text field displays a capital "L". The "L" means that the colormap uses a *local* range which allows you to modify the range without affecting the range of the same colormap used in other modules. By pressing the button you can toggle between *local* and *global* range.

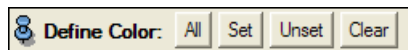
The default colormap has a constant color over the whole range. By leaving it constant you give all atoms the same color, which may be useful if you want to compare different molecules. For more information, see the section on *colormaps*.

Discrete CM



This colormap is used to map values of discrete attributes, like integers and strings, to colors.

Define Color



This port can be used to overwrite the standard colors of the selected color scheme. With the *All* button you can set a new color for all atoms. The *Clear* button unsets the user-defined colors for all atoms. The *Set* and *Unset* buttons operate on the set of highlighted atoms. These buttons can be used to set and unset the colors of those atoms, respectively.

6.3.2 Selecting and Filtering atoms

Various tasks require the selection of atoms in a molecule. The most prominent are the alignment of molecules and the selective display of molecule parts. *amira* offers two selection methods. You can select atoms visually via any of the viewing modules. Alternatively, selection can be done via the molecule's *selection browser*, which can be opened by pressing the *Show* button of the molecule.

6.3.2.1 Selection of Atoms with a Viewing Module

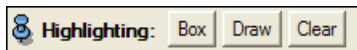
The selection of atoms with a viewing module can be done in two ways. The first way is by clicking on certain displayed parts of the molecule. In general, this will highlight the selected parts. By default, the number of atoms that will be affected by a click depends on the selected group level of the *color port*. For example, if the atoms in the viewing module are colored according to the *residues* level, all atoms belonging to the same residue as the picked atom will get selected. However, since this is very restrictive, you can easily change the selection mode to the most common levels, i.e., atoms, residues, secondary structures, and chains. In order to choose a certain level for the selection you need to press certain keys in advance. **Ctrl-a** chooses the *atoms* level, a click on an atom will only influence the selection for this atom. **Ctrl-r**, **Ctrl-c** and **Ctrl-s** choose *residues*, *chains* and *secondary_structure* levels, respectively. To switch back to the default behavior, where coloring determines selection, press **Ctrl-d**.

If you pick another part, the previously highlighted atoms will be unhighlighted and the newly selected atoms are highlighted. **Ctrl-clicking** a part of the molecule leaves the existing selection state of all atoms unchanged except for the newly selected atoms. If the selected atoms had been selected before, they will get deselected, otherwise they will get selected. On the one hand, this allows you to add atoms to the selection, on the other hand it also allows you to deselect atoms.

If you select some group and afterwards select a second one from the same level holding down the **Shift**-key all groups between the two will also be selected. The **Shift**-key can also be used in conjunction with the **Ctrl**-key.

If you do any selection by clicking in the viewer there will be some output in the console window informing you about what you have selected, the amount of output can be customized via the *Preferences* dialog, which is opened by choosing the *Preferences* entry from the *Edit* menu. The preferences concerning *Molecular Pack* are found on the *Molecules* tab.

Highlighting



The second way to select parts of the molecule is by using the functionality of the *Highlighting* port.

- If you press the *Box* button, the molecule's bounding box will be displayed. All atoms contained in the box will be highlighted. You can change the size of the box in *interaction mode* by clicking on the highlighted handles (usually in a light green) of the box. Keep the mouse button pressed

and drag in the desired direction. Notice that the box will not exceed the molecule's bounding box. You can also move the box within the bounding box by clicking on one of the box's sides. Pressing the *box* button a second time hides the box.

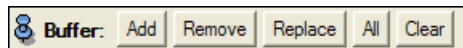
- After pressing the *Draw* button, you can draw a line in the viewer window which selects all atoms that are inside the line. Pressing the **Ctrl**-key while drawing inverts the selection. With the **Shift**-key you can remove atoms from the selection.
- The *Clear*-key deselects all atoms and hides the box if it is visible.

6.3.2.2 Filtering atoms

Filtering atoms gives us the ability to hide parts of the molecule for a certain module. All viewing modules and some computational modules, such as the *CompMolSurface* module, contain a *filter* that keeps track of all atoms of a molecule which are currently *in use*. The term *in use* means that the module will only see the *in use* atoms and regard all other atoms as non-existing. Thus, a viewing module will only display the *in use* atoms, and the computational module *CompMolSurface* will compute the molecular surface ignoring the *not in use* atoms. Each filter will register itself at the *molecule's selection browser*, so that you can easily determine the *in use* atoms of a module. For more information about how to do this, see the description of the *selection browser*.

In most modules, the filter's *Buffer* port will be visible, which adds to the convenience. The functionality of the *Buffer* port is described below.

Buffer



Pressing *Add* will append the selected atoms to the *in use* atoms. The *Remove* button will delete the selected atoms from the *in use* list. *Replace* will first clear the *in use* list and then add the selected atoms to it. Finally, the *All* and *Clear* buttons are shortcuts to delete or add all atoms from or to the *in use* list, respectively.

6.4 Aligning Molecules

To compare the structures of several molecules it is necessary to bring them into a suitable geometric arrangement relative to each other, because their absolute positions in the common coordinate system are generally meaningless. By arranging the molecules in various different ways, you can investigate various aspects.

6.4.1 Alignment of Trajectories

Molecular Pack offers a reusable component for the alignment of molecules. It appears in several modules, e.g., *Molecule*, *ConformationDensity* and *MeanMolecule*. Here, we will give an overview of the available functionality.

Two connection ports are supplied:

AlignMaster [optional]

To compute an alignment relative to a reference molecule, this port must be connected to the reference.

PrecomputedAlignment [optional]

This port can be connected to an alignment precomputed by using the module *PrecomputeAlignment*.

The control of the alignment is done via three ports, as described below:

Transformation



This port allows you to specify how the molecule should be aligned. The possible options are:

none: Atom coordinates are left as they are.

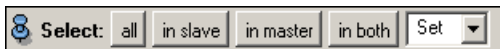
center of gravity: The molecule is positioned to the coordinate center (center of gravity) by applying a translation.

align to master: This mode requires a master molecule to be connected to the *AlignMaster* port which is used as a reference to which other molecules, the *slaves*, are aligned. We consider correspondences between atoms from the reference molecule, i.e. *master*, and atoms from the molecule to be aligned, i.e. *slave*. The alignment is done by minimizing the sum of squared distances between all corresponding atoms.

precomputed: This option is only enabled if the *PrecomputedAlignment* connection port is connected to an alignment object that has been previously computed. If *precomputed* is chosen, the transformation will be taken directly from this object.

external: This option indicates that the molecule's global transform has been explicitly set, e.g., via the Tcl command *setTransform* or by using the *Transform Editor*.

Select



This port becomes important if *align to master* is selected. It gives control over the atoms in the slave and the master that are used for the alignment. In the general case master and slave molecules are different, i.e. their numbers of atoms differ, and an alignment requires an explicit specification of pairs of atoms. This can be done by highlighting atoms in both molecules via the *MoleculeView* module and then pressing the *in both* button. Depending on whether the option menu is set to *Set* or *Add*, the

highlighted atoms will either replace an existing list of selected atoms or otherwise will be appended to it. The matching between atoms in the two molecules is defined by their order of selection.

The buttons *all*, *in slave*, and *in master* offer shortcuts for the frequent case of slave and master having the same number of atoms including the assumption that their order defines a natural matching. The easiest way is to use all atoms by pressing *all*. Selection of a subset can be done by highlighting in either the slave or the master and then pressing *in slave*, or *in master*, which will use the selected atoms in the respective molecule for the other molecule, too.

Selection



If *align to master* is selected, this port displays the existing state of selection that is used to compute the alignment. Possible forms are:

empty: No atoms are selected. At least three are necessary for an alignment.

all atoms: Master and slave have the same number of atoms and all are used for the alignment.

0, 5, 6: A list of indices implies that master and slave have the same number of atoms and identical subsets have been selected.

0 → 7, 5 → 6, 6 → 5: A list of index pairs (master index → slave index) indicates a matching resulting from individual selections in master and slave.

6.4.2 Mean Distance Alignment

Apart from the above mentioned alignment procedure, there exists a module that allows you to align two molecules by using the *mean distance* criterion instead of the *mean squared distance*. See *AlignMolecules* for more information.

6.4.3 Sequence alignment

Molecular Pack allows you to *align the sequences* of two molecules. Three algorithms are available for use depending on the kind of data and your needs: local, semi-global, and global alignment. The algorithms work both for proteins and ribonucleic acids, whereas t-RNA molecules are currently not supported because they contain modified bases. This module can be very helpful when used in conjunction with the *AlignMolecules* module.

6.5 Visualizing Molecular Trajectories and Metastable Conformations

A *MolTrajectory* can be visualized via animation of single time steps by attaching a *Molecule* module to the *MolTrajectory* and then attaching a *MoleculeView* or *BondAngleView* to the *Molecule*. The animation is controlled via the *Molecule's* *Time* port.

For *MolTrajectories* that represent metastable conformations, the modules *MeanMolecule*, *PrecomputeAlignment*, and *ConfigurationDensity* can be used. The *MeanMolecule* module aligns all steps of a trajectory and computes a mean molecule by averaging every atomic coordinate over all time steps. Instead of computing the mean molecule to a reference, as with the *MeanMolecule* module, the *PrecomputeAlignment* module allows you to find the optimal transformation of each time step to minimize the overall sum of squared distances. With the *RankTimeStep* module you can search in a trajectory for a desired time step, using different criteria, such as the *rmsd* value to a given reference.

The *ConfigurationDensity* module gives an impression of the fuzziness of the conformation by computing a probability density for the positions of atoms and bonds within a molecular trajectory. This density can then be visualized with the *Isosurface* and *Voltex* modules.

6.6 Atom Expressions

6.6.1 Overview

Atom expressions are a query language to find and select atoms of certain properties in the molecule for further action. The most important application of atom expressions in *amira* is the highlighting section of the *Selection Browser*.

In *amira*, a molecule is separated into groups of different levels. Each group contains a set of attributes. (More details of this concept can be found in the description of the *Attribute Editor*). Atom expressions are a simple form of a relational query language which accesses these attributes.

The simplest form of an atom expression is an *atom specifier*. This is a literal defining a level, one of its attributes, and a condition for this attribute. For example, `atoms/atomic_number=8` defines all atoms whose atomic number attribute equals 8 (i.e., all oxygens).

Such atom specifiers can be combined with logical operators like AND, OR and NOT. For example, `atoms/atomic_number=8 AND NOT atoms/charge=0` will select all charged oxygen atoms.

There are additional operators like WITHIN, BONDED and GROUP which apply certain conditions to coordinates or bond structure. These are explained in section on *operators*.

6.6.2 Grammar

All possible syntaxes of atom expressions are shown in the following grammar. The different literals and operators are further explained in the following sections.

```
atomExpr →      ( atomExpr )
                | NOT atomExpr
                | atomExpr AND atomExpr
                | atomExpr OR atomExpr
                | WITHIN (atomExpr,float)
                | BONDED (atomExpr[,int])
                | GROUP (groupParts)
                | CS
                | atomSpecifier
atomSpecifier →  hierName/[attrName=]ID
groupParts →     groupPart
                | groupPart,groupParts
groupPart →      groupSpecifier
                | groupSpecifier [bondOrderSymbolChar] groupSpecifier
groupSpecifier → [!] elementSymbol index
```

6.6.3 Literals

As mentioned in the overview, the simplest form of an atom expression is an atom specifier. An atom specifier consists of three literals: *hierName*, the optional *attrName*, and *ID*.

hierName stands for a name of a hierarchy level (e.g., *residues*). The following abbreviations can be used for the most common levels:

a=atoms, r=residues, b=bonds, s=secondary_structure, c=chains

attrName is optional and specifies the name of an attribute (e.g., *temperature*, *occupancy*, *type*, ...) of the given level. If it is omitted, the ID is assumed to specify the attribute name or index as shown by the list command. If an attribute name is given, the ID is assumed to stand for values of the attribute.

To see which hierarchy levels and respective attributes are defined for a given molecule, take a look at the *Color port* which is used in several modules. The right pull-down menu will show all available attributes for the level chosen in the left pull-down menu.

ID specifies an identifier of a member of the given hierarchy and attribute. It can be the name/index of the group (e.g., L112 for residue 112 on chain L) or the type (e.g., ASP for all aspartate residues). Wildcards such as * and ? may be used, or ranges (e.g., L112-L115) may be given. For the atom level there is a special case to be considered. Molecules in *amira* usually contain an atomic number attribute instead of an element symbol attribute. To select atoms via their element symbol you can simply type *a/element*. Thus the atom specifier for all oxygen atoms *a/O* is equivalent to the atom specifier *a/atomic_number=8*.

Instead of the '=' comparison you can also use the comparison operators '<', '>', '>=' and '<='. Note however that they are only available for index, integer and float attributes, not for string attributes.

Another atom expression form involving several literals is the *groupParts* expression which is used with the GROUP operator. Operators will be explained in the next section.

6.6.4 Operators

Logical Operators

Several *atomSpecifier* combinations can be used in one expression by linking them logically via the operators AND, OR, and NOT (&, |, and !). Priorities can be specified using usual parentheses (and).

WITHIN(*atomExpr*, *float*)

This operator selects all atoms which are nearer than *float* Å to any of the atoms specified by *atomExpr*.

BONDED(*atomExpr*[, *int*])

With this operator, all atoms that are recursively connected to any atoms specified by *atomExpr* will be chosen. You can optionally specify an integer value defining the maximal bond steps. If this is omitted, there will be no limit.

CS

CS specifies all currently selected (highlighted) atoms.

GROUP(*groupParts*)

The GROUP operator is a powerful tool to find functional groups by searching for a certain atom and bond pattern in the molecular graph. To define a graph as a search pattern (*groupParts*), you must divide it into linear pieces of sequential atoms (a *groupPart*). Each atom must be defined by its element symbol and an index which distinguishes it from other atoms with the same symbol but other indices (*groupSpecifier*). Thus, C1C2C3 would be a *groupPart* consisting of three different *groupSpecifiers* representing a chain of three carbons. This group part can now be combined with another group part by using one of its *groupSpecifiers* as branch point (e.g., C2O1H1 for a hydroxyl group branching from the second carbon of the chain). At the beginning of each group specifier, there can be an optional '!'. If it is given, the *groupSpecifier* is only used for matching, but the corresponding atoms will not be selected. (Thus !C1O1H1 would find the hydroxyl groups without selecting the flanking carbon, which must be given, however, to avoid selecting structures, such as OH groups in H2O). If the *atomSpecifier* in question appears several times (to define branch points), it is sufficient to mark it with the exclamation mark once.

Consecutive *atomSpecifiers* in a *groupPart* are usually not divided from each other. This means that there must be a bond of any type between the consecutive atoms. If you want to define the bond order further, you can give an optional *bondOrderSymbol*. ('-' for a single, '=' for a double, '#' for a triple and 'c' for an aromatic bond).

Take a carboxyl group as an example. To define the group pattern, the central carbon atom (C2) needs to be connected to three atoms: two oxygens (O1, O2), and one other carbon (C1). This can be done in the following way: GROUP(!C1C2O1 , C2O2H1). Thus, the hydroxyl group (O2H1) is given as a

branch of the CCO chain. There are, of course, several other ways to split this branch into linear pieces which you can easily find yourself. If your molecule contains the bond type attribute, you can also make use of the double bond. Thus the expression becomes `GROUP (! C1-C2=O1 , C2-O2-H1)`.

Notice: The keywords do not need to be in capital letters. Lower case letters, even a combination of lower and upper case letters, works as well.

6.6.5 Shortcuts

Molecular Pack also provides pre-defined shortcuts that have been assembled using the previously mentioned syntactical elements. The shortcuts can be found in your local `amira` directory in `share/molecules/atomExpr.cfg`, and can be edited and supplemented.

The standard aliases included in the current `amira` release are listed in Table 6.1.

6.6.6 Further Examples

- `atoms/5-8`
all atoms whose index is in the range 5 to 8, inclusive.
- `atoms/atomic_number>1`
all atoms, except hydrogens
- `s/type=helix AND NOT (a/C OR a/N)`
all atoms which belong to helices, except C and N atoms.
- `r/type=A*`
all atoms which belong to residues whose type name begins with the letter A.
- `BONDED(a/4 OR a/100,6)`
all atoms which are connected via at most 6 steps to the two specified atoms
- `WITHIN(r/A11,3.1) AND C`
all carbon atoms which are not away further than 3.1 angstroms from atoms of residue 11 on chain A
- `GROUP(C1C2C3C4C5C6C1)`
all cycles consisting of 6 carbons (e.g., cyclohexane).
- `acidic AND helix`
all atoms of acidic amino acids which belong to helices

acidic	acidic amino acids
acyclic	acyclic amino acids
aliphatic	aliphatic amino acids
alkali	atoms which are alkali metals
alkaliearth	atoms which are alkali earth metals
all	selects everything
amino	amino acids
aromatic	aromatic amino acids
at	adenine or thymine
backbone	atoms of protein or DNA/RNA
basic	basic amino acids
buried	amino acids which are usually found inside the protein
cg	cytosine or guanine
charged	charged amino acids
cyclic	cyclic amino acids
h2o	water molecules
helix	helices
halfmetallic	atoms which have half metallic properties
halogens	halogenic atoms
hetero	heterogenic atoms
hydrophobic	hydrophobic amino acids
ions	charged heterogenic atoms
metallic	atoms which have metallic properties
neutral	neutral amino acids
noble gas	atoms which have noble gas properties
nonmetallic	atoms which have nonmetallic properties
nucleic	nucleic acids
nucleicbackbone	backbone atoms of RNA/DNA
polar	polar amino acids
proteinbackbone	backbone of a polypeptide
purine	adenine or guanine
pyrimidine	cytosine or thymine
sheet	sheets
sidechain	atoms of the protein/RNA/DNA which do not belong to the backbone
site	
surface	amino acids which tend to be found on the surface of molecules
turn	

Table 6.1: Predefined expressions for selecting parts of molecules in amira.

Part III

VR Pack User's Guide

Chapter 7

VR Pack Configuration

VR Pack is an amira extension providing support for large tiled displays as well as immersive multi-wall displays like *CAVEs* or *Holobenches*. VR Pack supports multi-threaded rendering on multi-pipe machines, head tracking, active and passive stereo modes, advanced 3D user interaction, soft edge blending and many more. Any VRCO *trackd*-compatible tracking system can be used together with VR Pack. Existing amira modules can be directly used in an immersive environment by means of 3D menus. In addition, a simple API is provided, allowing an **Developer Pack** programmer to add display modules with a specific interaction behavior. Beginning with version 4.1 VR Pack can also be run on a graphics cluster. The particular requirements and limitations of this cluster version are described in a separate section below.

The documentation of VR Pack is separated into the following parts:

- *VR Pack essentials*
- *Flat screen configurations*
- *Immersive configurations*
- *Calibrating the tracking system*
- *The VR Pack cluster version*
- *Service management*
- *3D user interaction*
- *User-defined 3D-menu items*
- *Writing VR Pack custom modules*
- *AmiraVR control module reference*
- *Configuration file reference*
- *AmiraVR tutorials and demos*

7.1 VR Pack essentials

VR Pack can be configured in many different ways. A particular VR Pack configuration is described in a config file under `$AMIRA_ROOT/share/config` or `$AMIRA_LOCAL/share/config`. The config file contains things like the physical extent of the screens of the display system, information about the X display or the parts of the desktop mapped onto the screens, as well as calibration data for the tracking system.

When VR Pack is installed the `amira` main window provides an additional menu labeled *Config*. This menu lists all config files found in the config directory. Once a particular configuration is selected, an *VR Pack* module is created (if there not already exists one). Depending on the particular configuration the VR Pack module allows one to connect to the tracking system, to calibrate the tracking system, and to activate additional options.

The trackd server

VR Pack makes use of the VRCO *trackd* software in order to access the tracking system. However, *trackd* itself is not part of VR Pack. It has to be purchased and installed separately. For more information about this product please refer to `www.vrco.com` or `www.mc.com/tgs`.

Before a tracking system can be used in VR Pack the *trackd* server has to be started. The server connects to the tracking system and provides the actual tracker and controller data in two shared memory segments, which are read by `amira`. In contrast to previous versions of VR Pack, it is no longer necessary to manually link additional libraries like `libtrackdAPI.so` into the `amira lib` directory. Once the *trackd* server is running it can be accessed automatically.

Starting amira

After VR Pack has been installed `amira` can be started as usual. However, in order to activate parallel rendering of screens assigned to different thread groups, `amira` should be started with the `-mt` command line options. This option enables multi-threading. In order to permanently activate multi-threading, the environment variable `AMIRA_MULTITHREAD` can be set. In order to disable multi-threading when `AMIRA_MULTITHREAD` is set, `amira` can also be started with the `-st` command line option (single-threaded mode).

Note: On SGI Onyx systems there are known problems with the OpenGL driver related to the use of texture objects in different OpenGL contexts. If you are rendering more than one screen on a single pipe, you should define the environment variable `AMIRA_NO_CONTEXT_SHARING`. This bug will probably be fixed in IRIX 6.5.17 and higher.

7.2 Flat screen configurations

A flat screen configuration consists of usually two or more screens forming a bigger 2D virtual graphics window commonly called a *tiled display* or *power wall*. Users can interact with the ordinary 2D mouse, i.e., mouse events in the different windows are translated and interpreted in the 2D virtual window. There is no need for a tracking system in case of a flat screen configuration. For such a configuration

the only option provided by the VR Pack control module is a stereo toggle allowing the user to enable or disable stereo viewing. Besides standard OpenGL active stereo modes passive stereo modes can also be configured. In the most simple case this is done by defining two full screen windows on two different channels, one for the left eye view and one for the right eye view. This particular configuration is illustrated in Figure 7.1. Other passive stereo configurations are possible too, e.g., a super-wide tiled passive stereo configuration with four channels (left side left eye, left side right eye, right side left eye, right side right eye), possibly with an overlap region for soft-edge blending.

The main advantage of a flat screen configuration is its ease of use. There is no need for a tracking system. Flat screen configurations are well suited for presentations targeted to a larger audience, e.g., presentations in a seminar room or in a lecture hall.

In the following we describe some common flat screen configurations in more detail, namely a standard-resolution two-projector passive stereo configuration, a super-wide two-projector mono configuration with soft-edge blending, and a tiled 2x2 four-channel monitor configuration. For some cases also hardware solutions are available, namely special-purpose video splitters converting an interlaced active stereo signal into separate non-interlaced left eye and right eye signals for the passive stereo case, or hardware edge-blending units for blended super-wide configurations. However, these hardware solutions are usually expensive and less flexible. Therefore they are often not suitable for temporary demonstrations or experiments.

- *A two-channel passive stereo configuration*
- *A super-wide configuration with soft-edge blending*
- *A tiled four-channel 2x2 monitor configuration*

7.2.1 Example: A two-channel passive stereo configuration

For a single-screen passive stereo projection system two video projectors emitting orthogonally polarized light are required. One projector displays the left eye image, the other one the right eye image. This is illustrated in Figure 7.1. Both images are projected onto a non-depolarizing screen either using front projection or rear projection. Observers wear suitable light-weight polarized glasses so that each eye only sees its own image, but not the image determined for the other eye. Without special-purpose hardware such a passive stereo projection system can be driven in full screen mode even using a very inexpensive low-end dual-head graphics adapter (for example NVidia GeForce2 MX TwinView).

We assume that the dual-head graphics computer is configured so that the left half of the desktop is output on one channel and the right half is output on the other channel. Here is the VR Pack configuration file:

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name                "Left eye view"
```

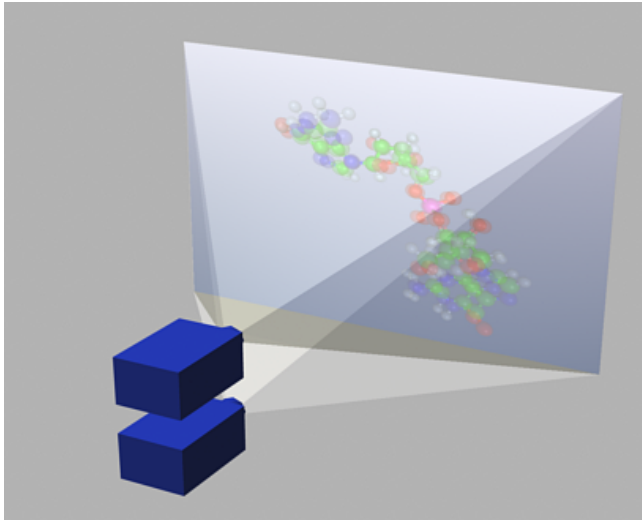


Figure 7.1: Sketch of a single-screen passive stereo projection system.

```

channelOrigin    0 0
channelSize      0.5 1
tileOrigin       0 0
tileSize         1 1
cameraMode       LEFT_VIEW
}
SoScreen {
  name            "Right eye view"
  channelOrigin   0.5 0
  channelSize     0.5 1
  tileOrigin      0 0
  tileSize        1 1
  cameraMode      RIGHT_VIEW
}
}

```

The fields *channelOrigin* and *channelSize* indicate that the two windows exactly cover the left half and the right half of the desktop. The fields *tileOrigin* and *tileSize* indicate that both screens should display the full viewer window, i.e., there is no tiling at all. Finally, the field *cameraMode* indicates that one screen should display the left eye view and the other the right eye view. Note that the graphics computer need not to support active stereo for this configuration. In order to change the default stereo parameters *eye offset* and *stereo balance* the following standard *amira* Tcl command can be used:

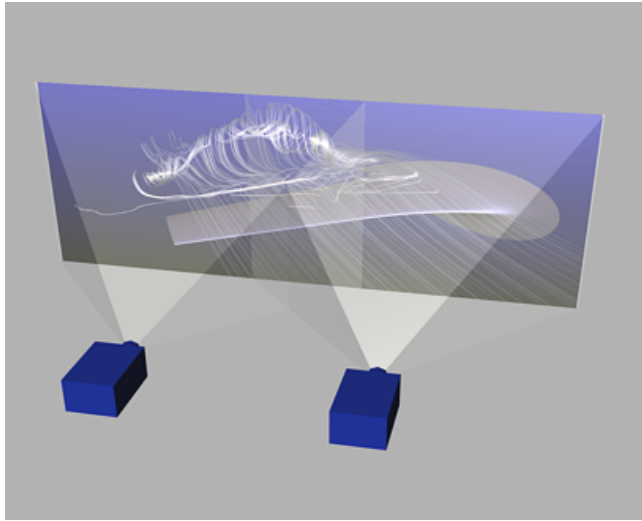


Figure 7.2: Sketch of a super-wide projection system with soft-edge blending.

```
viewer 0 setStereo [-b <balance>] <offset>
```

Here `<offset>` denotes the eye offset and `<balance>` denotes the stereo balance, i.e., the location of the zero parallax plane. Depending on the balance value objects appear to be in front of the projection screen or behind it.

7.2.2 Example: A super-wide configuration with soft-edge blending

Super-wide images with two times the XGA- or SXGA-resolution can be displayed using two projectors and a low-end dual-head graphics adapter (for example NVidia GeForce2 MX TwinView). However, often it is very desirable to have an overlap between the two projected images. In the overlap region one image softly fades out from full intensity to black, while the other image fades in from black to full intensity. This technique is called *soft-edge blending* (compare Figure 7.2). With soft-edge blending the border between the two projected images becomes almost invisible. VR Pack is able to generate two partially overlapping images. In addition, the soft-edge can be computed in software. With these features perfect full-screen demos can be presented on a super-wide projection system.

We assume that the dual-head graphics computer is configured so that the left half of the desktop is output on one channel and the right half is output on the other channel. We further assume that there is a 20 percent overlap between the projected images of the two projectors. Here is the VR Pack configuration file:

```
#Inventor V2.1 ascii
```

```

Separator {
    SoScreen {
        name            "Left half"
        channelOrigin    0 0
        channelSize      0.5 1
        tileOrigin       0 0
        tileSize         0.6 1
        softEdgeOverlap  [ 0, 0.2, 0, 0 ]
        softEdgeGamma    [ 0, 1.2, 0, 0 ]
    }
    SoScreen {
        name            "Right half"
        channelOrigin    0.5 0
        channelSize      0.5 1
        tileOrigin       0.4 0
        tileSize         0.6 1
        softEdgeOverlap  [ 0.2, 0, 0, 0 ]
        softEdgeGamma    [ 0, 1.2, 0, 0 ]
    }
}

```

The fields *channelOrigin* and *channelSize* indicate that the two windows exactly cover the left half and the right half of the desktop. The fields *tileOrigin* and *tileSize* indicate that both screens display an area of 0.6 times the width of the full tiled window. The first screen displays the left half of that window, the second screen displays the right half. The field *softEdgeOverlap* specifies the relative width of the soft-edge region at the left, right, bottom, and top border of the screen. For the first screen there is a 20 percent soft-edge region at the right border, for the second screen there is a 20 percent soft-edge region at the left border. Finally, the field *softEdgeGamma* specifies the gamma factor for the soft-edge region. The gamma factor determines how the fading from full intensity to black is done. A gamma factor of one means a linear transition in terms of RGB values. However, since RGB values are usually not mapped linearly to light intensity by the video projector often a decrease of overall light intensity is observed in the overlap region. In order to compensate for this a gamma factor larger than one can be used.

In order to facilitate the initial calibration of a super-wide projection system with soft-edge blending the VR Pack module provides a special-purpose Tcl command *setSoftEdge*. This command is used in the following way:

```
AmiraVR setSoftEdge -g <gamma> <overlap> [<overlap>...]
```

This command automatically creates a two-screen configuration similar to the one above. However, the correct values for *tileSize*, *tileOrigin*, and *softEdgeOverlap* are computed automatically from the

relative overlap value. By gradually adapting this value the correct settings for a given but unknown setup of the projection system can be easily found.

7.2.3 Example: A tiled four-channel 2x2 monitor configuration

There are some interesting graphics workstations with two two-channel graphics pipes. One example is the SGI Octance Fuel. With such a computer four different monitors or projectors can be used. For some purposes it is useful to have a single viewer subdivided into 2x2 parts and each part being displayed by a different monitor. Such a configuration can be easily created using VR Pack.

We assume that the graphics computer has two pipes with two channels each. The two pipes are configured as two separate X11 screens, :0.0 and :0.1. The left and right part of each screen is output on the two different channels of the particular pipe. Then the configuration files looks as follows:

```
#Inventor V2.1 ascii

Separator {
  SoScreen {
    name          "Upper left monitor"
    display        ":0.0"
    channelOrigin  0 0
    channelSize    0.5 1
    tileOrigin     0 0.5
    tileSize       0.5 0.5
    threadGroup    0
  }
  SoScreen {
    name          "Upper right monitor"
    display        ":0.0"
    channelOrigin  0.5 0
    channelSize    0.5 1
    tileOrigin     0.5 0.5
    tileSize       0.5 0.5
    threadGroup    0
  }
  SoScreen {
    name          "Lower left monitor"
    display        ":0.1"
    channelOrigin  0 0
    channelSize    0.5 1
    tileOrigin     0 0
    tileSize       0.5 0.5
```

```

        threadGroup      1
    }
    SoScreen {
        name              "Lower right monitor"
        display            ":0.1"
        channelOrigin 0.5 1
        channelSize 0.5 1
        tileOrigin         0.5 0
        tileSize           0.5 0.5
        threadGroup       1
    }
}

```

Since the two X11 screens `:0.0` and `:0.1` are driven by two independent graphics pipes it makes sense to perform the rendering on these pipes in parallel. This is done by assigning the corresponding screens two different thread groups. Note that for all common current graphics architectures it makes no sense to render multiple windows on the same pipe in parallel. Typically, this even implies a significant performance decrease. Therefore, here we use only two thread groups instead of four. In order to actually activate parallel rendering **amira** must be started with the command line option `-mt`. Alternatively, the environment variable `AMIRA_MULTITHREAD` can be defined.

7.3 Immersive configurations

In **VR Pack** an immersive configuration differs from a flat screen configuration mainly in the way the screens are described in the config file. While for a flat screen configuration it was sufficient to specify which part of a big 2D virtual screen was covered by each screen, for an immersive configuration the true physical coordinates of the screens have to be specified. Knowing the exact spatial arrangements of the screens has the advantage that correct perspective views can be computed for all screens, provided the 3D position of the observer is also known. In particular, the different screens no longer need to be arranged in a plane. Instead, the screens might be arranged perpendicular to each other like in a *CAVE* or on a *Holobench*. In fact, **VR Pack** supports any other oblique arrangement as well.

Since the 3D position of the observer need to be known in order to compute correct perspective views usually the observer's eye position is tracked in an immersive environment. In **VR Pack** this so-called *head tracking* can be achieved using a variety of different tracking systems. Instead of accessing the tracking system directly an intermediate software layer is used, namely the *trackd* software of **VR CO**. *trackd* runs as a server, communicates with the tracking system, and stores the tracking data in a shared-memory areas which then is read by **VR Pack**. Using a second sensor not only the observer's eye position can be tracked, but also the position of a virtual wand, i.e., a kind of pointing or interaction device to be held in a hand. For large planar screen configurations it sometimes also makes sense to use a virtual wand without head tracking. The images then will always be computed for a fixed observer's eye position. Image flicker due to noise in the tracking data is avoided, which is an advantage for 3D

demonstrations in front of a larger audience.

Obviously, immersive configurations are more difficult to setup than flat screen configurations. Besides defining a suitable VR Pack configuration file, also the tracking system and the *trackd* server have to be properly initialized. Finally, the tracking system has to be calibrated for use with VR Pack. In the following sections we first want to present some example config files for common immersive environments. In particular config files for a single-wall workbench, for a double-wall Holobench, and for a four-sided CAVE shall be discussed. The process of calibrating the tracking system and customizing 3D user interaction is described in subsequent sections.

- A Workbench configuration
- A Holobench configuration
- A CAVE configuration

7.3.1 Example: A Workbench configuration

A single-screen 3D projection system is often called an immersive workbench. The actual projection screen can either be in up-right position, or it can be oriented like the surface of a table. Of course, any other orientation is possible too. There are even devices like the Barco Baron which can be arbitrarily tilted between fully horizontal and fully vertical position. For VR Pack there is no difference between these configurations, as long as the tracking system is calibrated in the right way. Traditionally a workbench is driven by a CRT projector using active stereo. However, today also passive stereo systems based on two LCD or DLP projectors become more common. In the config file below we assume that an active stereo system is used, or that a passive stereo system is connected via an appropriate signal splitter. This means that the workbench can be used with any standard stereo-capable graphics computer. No dual-head or multi-pipe computer is necessary. An example of a single-screen workbench is shown in Figure 7.3.

An VR Pack config file for driving a single-screen immersive workbench with a tracked 3D input device but without head tracking is listed below. Instead of actually tracking the observer's eye position in this case a fixed default camera position is used. This can be useful for demonstrations in front of small or medium-size groups, since it produces less fidget images. In addition, it saves a second sensor for the tracking system.

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name            "Workbench"
        lowerLeft        0 0 0
        lowerRight       170 0 0
        upperRight       170 130 0
        upperLeft        0 130 0
    }
}
```



Figure 7.3: A single-screen 3D projection system.

```

        cameraMode      ACTIVE_STEREO
    }
    SoTracker {
        server            "4147:4148"
        autoConnect       TRUE
        wandTrackerId     0
        headTrackerId     -1
        defaultCameraPosition 85 65 140
        defaultObjectPosition 85 65 0
        referencePoints [ 0 0 0, 170 0 0, 170 130 0, 0 130 0 ]
    }
}

```

In the *SoScreen* section of the config file the physical coordinates of the four corners of the workbench are defined. This can be done using an arbitrary right-handed coordinate system with arbitrary units. In this case the coordinates might be specified in centimeters. The origin of the coordinate system was chosen in the lower left corner of the screen. By default, a full-screen graphics window is opened when activating this configuration. The field *cameraMode* indicates that the graphics window should be opened in active stereo mode by default.

In the *SoTracker* section of the config file the tracking system is described. First the shared memory ids of the trackd server as specified in the *trackd.conf* file are listed. The syntax is *id of controller*

reader:id of tracker reader. The *autoConnect* field indicates that a connection to the trackd server should be established automatically as soon as the configuration is activated. *wandTrackerId* denotes the trackd sensor id of the 3D input device. Head tracking is disabled by setting *headTrackerId* to -1. Instead the default camera position set in the line below is used. This position must be specified using the same coordinate system as the screen. In this case the camera is located 140 cm in front of the screen's center. The default object position is the position where the scene is placed by default (or whenever a *view all* request comes from the viewer).

At the end of the config file four reference points are specified, namely the four corners of the screen. Before the configuration can be actually used, the tracking system has to be calibrated (see section 7.4). This is done by placing the input device at the reference points and clicking an input button. Once the tracking system is calibrated, the config file should be written by clicking the *write config* button of the VR Pack control module. The new config file will contain the same information listed above, but in addition it will also contain some calibration data, e.g., the transformation between raw tracker coordinates and screen coordinates.

7.3.2 Example: A Holobench configuration

A Holobench (TM) is a special display system consisting of two screens oriented perpendicular to each other. One screen is oriented vertically, the other one is oriented horizontally like a table. On a good Holobench there is almost no visible border between the two screens. Provided the observer's eye position is known, correct perspective views can be computed so that the displayed scene finally does not appear to have any break. However, in contrast to a single-screen workbench this is only the case for the tracked observer. Other spectators will more or less clearly notice a break.

In order to drive a workbench at least a stereo-capable dual-head graphics computer is required. In principle, two different settings are possible. Either, there is a big desktop and one half of it is output on a first channel and the other one is output on a second channel. Or, there are two independent graphics adapters (pipes) which are configured as two different X11 displays or as one display with two X11 screens. In the config file below we assume, that the desktop is split into two halves (left and right). An example of how to use a multi-pipe graphics computer is presented in the next section where a 4-sided CAVE configuration is described.

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name            "Vertical Screen"
        lowerLeft        0 0 0
        lowerRight       180 0 0
        upperRight       180 110 0
        upperLeft        0 110 0
        channelOrigin    0 0
```

```

        channelSize      0.5 1
        cameraMode       ACTIVE_STEREO
    }
    SoScreen {
        name              "Horizontal Screen (rotated)"
        lowerLeft         180 0 0
        lowerRight        0 0 0
        upperRight        0 0 110
        upperLeft         180 0 110
        channelOrigin     0.5 0
        channelSize       0.5 1
        cameraMode       ACTIVE_STEREO
    }
    SoTracker {
        server            "4147:4148"
        autoConnect       TRUE
        wandTrackerId     1
        headTrackerId     0
        leftEyeOffset     6 0 0
        rightEyeOffset    13 0 0
        defaultCameraPosition 90 55 110
        defaultObjectPosition 90 20 20
        referencePoints [ 60 0 110, 120 0 110, 120 0 55, 60 0 55 ]
    }
}

```

In the two *SoScreen* sections of the config file the geometry of the Holobench is described by specifying the physical coordinates of the four corners of each screen. An arbitrary right-handed coordinate system with arbitrary units can be chosen. Here the coordinates are specified in centimeters and the origin was put in the lower left corner of the vertical screen. Notice, that the horizontal screen was rotated by 180 degrees with respect to the vertical screen. I.e., instead of the upper left corner the lower right corner of the horizontal screen is located at the origin. This is because the horizontal image of a Holobench is usually projected that way. If the corresponding lower scan-lines of the two images meet at the border between the two screens artifacts due to delayed response of the active shutter glasses are avoided. The fields *channelOrigin* and *channelSize* indicate that the graphics window for the vertical screen should be opened on the left half of the desktop, while the graphics window for the horizontal screen should be opened on the right half. Both windows are opened in stereo mode by default as specified by *cameraMode*.

In the *SoTracker* section of the config file the tracking system is described. First the shared memory ids of the trackd server as specified in the *trackd.conf* file are listed. The syntax is *id of controller reader:if of tracker reader*. The *autoConnect* field indicates that a connection to the trackd server should be established automatically as soon as the configuration is activated. *wandTrackerId* denotes the trackd

sensor id of the 3D input device. *headTrackerId* denotes the trackd sensor id of the head sensor which is usually mounted at the shutter glasses. The fields *leftEyeOffset* and *rightEyeOffset* specify the actual position of the eyes with respect to the head sensor. Standing in front of the Holobench the x-axis points horizontally to the right, the y-axis points upwards, and the z-axis points towards the observer. In this case we assume the head sensor to be mounted on the left side of the glasses since both left and right eye have a positive offset in x-direction. The default camera position and the default object position both are specified in the same coordinate system as the screens. The default camera position is only used if the tracking system is disconnected. The default object position is the position where the scene is placed by default (or whenever a *view all* request comes from the viewer).

At the end of the config file four reference points are specified, namely four points on the horizontal screen. Before the configuration can be actually used, the tracking system has to be calibrated (see section 7.4). This is done by placing the input device at the reference points and clicking an input button. The reference points were chosen so that they can be easily accessed with the hand. In addition, it is important that the points are well inside the operating range of the tracking system. During calibration the raw coordinates of the wand sensor are displayed, so you can check if these values are reasonable. Once the tracking system is calibrated, the config file should be written by clicking the *write config* button of the VR Pack control module. The new config file will contain the same information listed above, but in addition it will also contain some calibration data, e.g., the transformation between raw tracker coordinates and screen coordinates.

7.3.3 Example: A 4-side CAVE configuration

A CAVE (TM) is a more or less fully immersive VR display system with the shape of a cubical box. Typically the size of the box is something like 3 x 3 x 3 meters. Three, four, five, or even six sides of the box are implemented as projection screens. In this way an observer inside the box can completely dive into a virtual world. In order to compute correct perspective views the eye position of the observer need to be tracked. Other non-tracked observers will perceive distorted images, especially at the edges between the individual walls. A schematic view of a 3-side CAVE is shown in Figure 7.4.

The VR Pack config file listed below was designed for a 4-side CAVE. In order to drive such a system four different images need to be generated, one for the front, bottom, left, and right wall respectively. This can be done for example using a two-pipe SGI Onyx system. Each pipe has two channels. Thus it is able to output two different images. We assume that there is one X server running on the machine. The two pipes are configured as two independent X11 screens, denoted :0.0 and :0.1. On each X11 screen there is a double-wide desktop. The left half and the right half of the two desktops are output by the two different channels of each pipe. For such a setting the VR Pack config file looks as follows:

```
#Inventor V2.1 ascii

Separator {
    SoScreen {
        name          "Front"
        display        ":0.0"
```

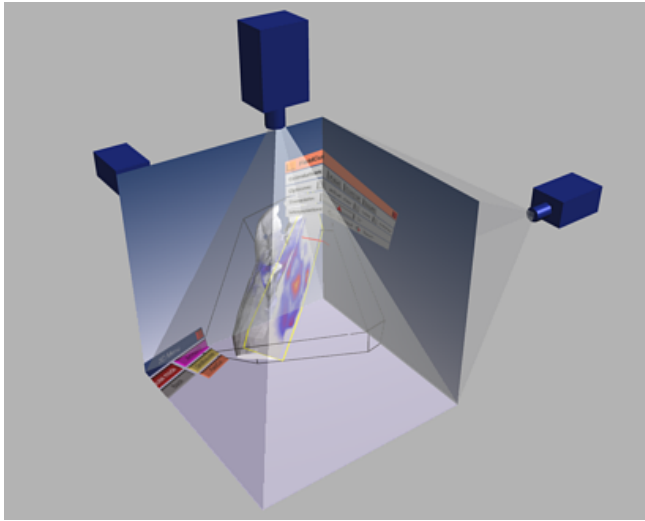


Figure 7.4: Schematic view of a 3-side CAVE system.

```

lowerLeft      0 0 0
lowerRight     300 0 0
upperRight     300 300 0
upperLeft      0 300 0
channelOrigin  0 0
channelSize    0.5 1
cameraMode     ACTIVE_STEREO
threadGroup    0
}
SoScreen {
    name        "Bottom"
    display     ":0.0"
    lowerLeft   0 0 300
    lowerRight  300 0 300
    upperRight  300 0 0
    upperLeft   0 0 0
    channelOrigin 0.5 0
    channelSize 0.5 1

    # If the bottom image is rotated try this:
    # lowerLeft     300 0 0
    # lowerRight    0 0 0
    # upperRight    0 0 300
    # upperLeft     300 0 300

    # This modifies the \amira gradient background
    backgroundMode BG_LOWER
    cameraMode     ACTIVE_STEREO

```

```

        threadGroup      0
    }
    SoScreen {
        name              "Left"
        display            ":0.1"
        lowerLeft          0 0 300
        lowerRight         0 0 0
        upperRight         0 300 0
        upperLeft          0 300 300
        channelOrigin      0 0
        channelSize        0.5 1
        cameraMode         ACTIVE_STEREO
        threadGroup        1
    }
    SoScreen {
        name              "Right"
        display            ":0.1"
        lowerLeft          300 0 0
        lowerRight         300 0 300
        upperRight         300 300 300
        upperLeft          300 300 0
        channelOrigin      0.5 0
        channelSize        0.5 1
        cameraMode         ACTIVE_STEREO
        threadGroup        1
    }
    SoTracker {
        server             "4147:4148"
        autoConnect        TRUE
        wandTrackerId      1
        headTrackerId      0
        leftEyeOffset      6 0 0
        rightEyeOffset     13 0 0
        defaultCameraPosition 50 50 100
        defaultObjectPosition 50 50 0
        referencePoints [
            0 150 100, 0 150 100, 100 150 0, 200 150 0, 300 150 100 ]
    }
}

```

7.4 Calibrating the tracking system

This section describes how to calibrate a tracking system for use in VR Pack. Here, calibration essentially means finding the transformation between the raw tracker coordinates and the coordinates in which the geometry of the display system, i.e., the corners of the screens, has been defined. Calibrating the tracking system does not involve changing the trackd config file *trackd.conf* in any way. Instead the calibration data is completely stored in the VR Pack config file. Usually the tracking system need to be calibrated only once. A new calibration is necessary for example if the antenna of an electromagnetic tracking system is moved with respect to the display system, if the screen of a single-wall immersive workbench is tilted, or if a new 3D input device is used where the 3D sensor is oriented in a different

way. The only part of the calibration process which one might repeat many a time is picking the wand. Picking the wand means to specify the offset between the actual position of the 3D input device and the visual representation of the wand in *amira*. Sometimes it is useful not to have any offset, while sometimes an offset provides more pleasant less exhausting working conditions. The wand offset can be quickly adapted even multiple times within a single VR Pack session.

Below there is a complete step-by-step description of the calibration process. Before starting calibration make sure that you have a valid VR Pack config file for your particular display system and that the trackd software and the tracking system itself are setup in the right way. In particular, make sure that the tracker coordinates are reported in a right-handed coordinate system. Many electromagnetic tracking systems cannot distinguish between two opposite hemispheres. In the *trackd.conf* file you have to specify in which hemisphere you want to operate. If the wrong hemisphere is specified a left-handed coordinate system is used, which cannot be correctly calibrated in *amira*.

1. Make sure that the trackd server is running and that the tracking system is operating. Start *amira* and choose the appropriate configuration from the main window's *Config* menu.
2. In the VR Pack control module connect to the tracking system if necessary. Activate the *values* toggle in order to display the current 3D coordinates in the upper left corner of the main screen. Make sure that the coordinates are changing if you move the 3D input device and the head sensor (unless head tracking is disabled in the VR Pack config file). Make also sure, that the button status changes if you press a button of the 3D input device.

Note: Unless you are in calibration mode the coordinates reported by the *values* option are transformed coordinates, i.e., they are in the same coordinate system in which the screens in the config file have been defined.

Hint: If the wand tracker and the head tracker seem to be exchanged modify the fields *wandTrackerId* and *headTrackerId* in the VR Pack config file.

3. Click the *Calibrate* button of the VR Pack control module. You are now in calibration mode. A fixed 2D control grid is displayed on all screens. You are requested to click at the first reference point. In the current version of VR Pack, the control grid is not guaranteed to match the actual reference points. However, you could choose the reference points in the config file to be at the corners of the control grid (one third and two thirds of the screen width in horizontal direction, one half of the screen height in vertical direction). Besides the number of the reference point its coordinates as specified in the config file are displayed in the upper left corner of the main screen.

Now move the 3D input device at the first reference point and click any button of the device.

Note: Make sure that the raw tracker coordinates which are displayed in the upper left corner of the main screen are valid at the reference point. Some devices just report zero values if the sensor is outside the operating range of the tracking system. Some other devices report non-sense values, typically with large oscillations. If you don't get stable values at a reference point, choose some other point in the VR Pack config file.

4. Next you are asked to click at second reference point using the 3D input device. Repeat the above procedure until all reference points have been located.

Note: If not at least three different reference points are specified in the config file, by default the upper left, lower left, and lower right corner of the first screen are used as reference points. You may want to specify other reference points or a larger number of reference points in order to improve accuracy. *The reference points must not lie all on one line.* However, they may well be located in a common plane, e.g., in the plane of the main screen.

5. Next, you need to align the glasses for head tracking. However, if head tracking is disabled in the VR Pack config file, this step is omitted.

Align the glasses parallel to the x-axis of the screen coordinate system. Usually the x-axis will be oriented horizontally with respect to the front screen of the display system. The up-direction of the glasses should be aligned parallel to the y-axis of the screen coordinate system. Usually the y-axis will be oriented vertically with respect to the front screen. Once you have aligned the glasses click any button of the 3D input device.

6. Now camera calibration is done and correct stereoscopic images should be displayed. As a final step you are requested to pick the wand in order to define the wand offset. The wand is being displayed half-way between the current eye position (or the default camera position, if head tracking is disabled) and the default object position. Now move the 3D input device near the origin of the wand (or at any other position) and click any button. The virtual wand remains stick to the 3D input device in exactly that position. You can repeat this last calibration step any time by clicking on the *pick wand* button of the VR Pack module.

Hint: If you don't get a clear 3D image of the wand make sure left and right eye images are not exchanged. If the eyes are exchanged you get a result which somehow also looks 3D but which isn't comfortable to work with at all. Also make sure that the fields *leftEyeOffset* and *rightEyeOffset* are set correctly in the config file.

7. At this point, the calibration is finished. You may now want to write a new VR Pack config file (or to overwrite the original one) in order to save the calibration data permanently. This can be achieved most easily by pressing the *write config* button of the VR Pack module. Pressing this button causes the *amira* file browser to be activated. You can accept the name of the previous config file or choose a new one.

When reloading the new configuration calibrated tracker data will be generated automatically. You can verify the correctness of the calibration process by turning on the *values* toggle and moving the 3D input device to one of the reference points. The reported values should be almost the same as the coordinates of the reference point specified in the config file.

7.5 The VR Pack cluster version

The VR Pack cluster version is an extension allowing VR Pack to be used on a graphics cluster. A graphics cluster consists of multiple computers connected via a network. Each computer controls one or more screens of a multi-wall display system. VR Pack synchronizes the different nodes, and ensures that same scene is rendered simultaneously from different perspectives. The VR Pack cluster version facilitates parallel multi-pipe rendering. It does not speed up ordinary computations by distributing

them across multiple nodes of the cluster. Instead, on each node a separate instance of **amira** is running, with the complete network and all data being duplicated. Changes of **amira** modules made on the master node will be propagated to the slave nodes and executed synchronously. A 2D mouse can be used to manipulate data on the master node.

Requirements:

- All data files and scripts must have the same absolute file path on all nodes of the cluster. Likewise, **amira** must be installed at the same location on all nodes. For convenience, it is recommended to run **amira** from a mounted file system. This ensures that the same scripts and config files will be used on all nodes.
- All nodes of the cluster should be of the same type. If different hardware is used, it is possible that the cluster will run out-of-sync because of round-off errors. Rendering speed is limited by the slowest node of the cluster.
- All nodes of the graphics cluster must be able to communicate with each other via a TCP/IP connection. A standard 10 Mbit ethernet connection is fast enough, since only synchronization commands are exchanged, rather than images or raw data blocks.
- The graphics cards of the different nodes must be genlocked with a genlock cable. Currently only a small number of graphics cards (such as WildCat or SUN Zulu boards) provide genlocking. Genlocking ensures that the video refresh cycles are synchronized. **VR Pack** itself only synchronizes OpenGL buffer swaps.

Limitations:

- The focus of the **VR Pack** cluster version is on working in a VR environment. Consequently all manipulations performed in VR mode, e.g., via the 3D menu, will be properly synchronized. On the other hand, currently not all manipulations made on the master node via the conventional 2D user interface will be synchronized.
- The general work flow is to first create a network in standard mode on the master node, and then to load this network from file in cluster mode. Currently, you cannot create new modules via an object's popup menu in cluster mode, nor you can interactively change connections via the 2D user interface.
- Some modules such as **PlanerLIC** or **DisplayISL** make use of random number generators. Currently, it is not ensured that the resulting numbers are always the same on all nodes. Consequently, different visual results might be obtained on different nodes.
- Any kind of 2D mouse interaction in a 3D viewer window also will not be synchronized in cluster mode. Thus you cannot draw a lasso area with the 2D mouse. Mouse manipulation and interaction are entirely disabled on slaves nodes.
- Editors and clip planes are not synchronized in cluster mode.
- Loading a new cluster config file removes all visible objects from the Pool.

Running VR Pack cluster

The following steps are required to run the VR Pack cluster version:

1. Install VR Pack on every node of the graphics cluster under the same absolute path name, or better yet, create a mounted file system with the same absolute path name on every node.
2. Create a standard VR Pack config file describing the geometry of your display system. Then add a field *hostname* in each screen section. This field indicates on which node the screen will be rendered. The config file must be stored on all nodes in the directory `$AMIRA_ROOT/share/config`.
3. Choose one node as the master node. After **amira** installation, a daemon is started by a service on Windows systems and `inetd` on Unix/Linux systems. Start VR Pack on the master and select your cluster configuration from the config menu. Slave instances of VR Pack should now be started on all nodes specified in the config file.

To use the old `-clusterdaemon` option, stop the daemon (see *service management*) and replace step 3 by the following steps:

1. Choose one node as the master node. On all other nodes start **amira** with the command line option `-clusterdaemon`. With this option a little daemon is started, to which the master process can talk. The daemon automatically starts a slave instance of the real **amira** if necessary.
2. Start VR Pack on the master node. Then select your cluster configuration from the config menu. Slave instances of VR Pack should now be started on all nodes specified in the config file.
3. Next you can load any standard **amira** network script. For example, open the online help browser on the master node and choose one of the standard VR Pack tracking demos. The particular script will be loaded on all slaves as well. Once a script has been loaded, all standard VR Pack interaction modes are available in cluster mode too.

7.6 The VR Pack service

Note: To manage the **amira** service, you must have administrator privileges (Windows) or root privileges (Unix/Linux).

How to stop, start, and restart the **amira service ?**

- On Windows platforms:

During system initialization, a Windows service is started (`amiraservice.exe`), which manages the daemon (`amiradaemon.exe`).

Service management is done through a Windows interface launched via the Control Panel (select Administrative tools, then Services) or with the following command: `services.msc /s`

- To start the daemon, select **amira** service in the list, then select Start from the Action menu.
- To restart the daemon, select **amira** service in the list, then select Restart from the Action

menu.

- To stop the daemon, select **amira** service in the list, then select Stop from the Action menu.

Note: To install or uninstall the service, run the `serviceinstall.exe` or `serviceuninstall.exe` programs located in the `%AMIRA_ROOT%/bin` directory.

Warning about virtual drives: The service does not support logical or network volumes. Indeed, such virtual drives do not exist in the service execution context. You must specify full paths constructed according to the universal naming convention (UNC), such as `\\remotemachine\directory`.

For example, to install the service from a virtual drive `S:` targeting a distant mount point `\\remotemachine\directory`, do not launch the installer from `S:`, but from `\\remotemachine\directory`. Then, for the same reason, the user should not load data from virtual drives because **amira** slaves cannot resolve virtual paths unless they are below `%AMIRA_ROOT%`, in which case data paths are re-interpreted on slaves with the local `%AMIRA_ROOT%`.

- On Unix/Linux platforms:

Mercury provides a full set of shell scripts in order to manage the **amira** daemon. They are located in the directory `$AMIRA_ROOT/setup`.

Two shell scripts, `install (serviceSetup.sh)` and `uninstall (serviceUninstall.sh)`, are available to cause the **amira** daemon to be launched or not during system initialization. They require `inetd` or `xinetd` to be installed on the machine. If neither of these services managers is available, the user will have to launch the **amira** daemon manually.

The installation script (`serviceSetup.sh`) can take the following parameters:

- `-daemonPath < PATH >`: with the path of the daemon.
- `-daemonExePath < PATH >`: with the path of the main application.

Remark: Do not use paths with space characters.

Example:

```
$ cd setup
$ sh serviceSetup.sh -daemonPath /opt/Mercury/Amira40/bin/amiradaemon
-d daemonExePath /opt/Mercury/Amira40/bin/start
```

If `inetd` is running, the **amira** service is added in the `inetd.conf` file:

```
amira STREAM TCP nowait root /opt/Mercury/Amira40/bin/amiradaemon
-d daemonExePath /opt/Mercury/Amira40/bin/start
```

Then the script launches `/etc/init.d/inetd`.

If `xinetd` is running, the **amira** service file is added in the directory `/etc/xinetd.d`:

```
service amira
{
```



```

type = UNLISTED
socket_type = STREAM
protocol = TCP
wait = false
user = root
server = /opt/Mercury/Amira40/bin/amiradaemon
server_args = -daemonExePath /opt/Mercury/Amira40/bin/start
port = 17234
}

```

Then the script launches `/etc/init.d/xinetd`.

Two more scripts are provided for convenience (`installCluster.sh` and `uninstallCluster.sh`) to set up the `amira` service over all cluster nodes. Those scripts take as parameters the nodes to which they attempt to connect through `ssh`.

Example:

```

$ cd setup
$ sh installCluster.sh node1 node2 node3 node4

```


Chapter 8

VR Pack Interaction

8.1 3D user interaction

VR Pack flat screen configurations, i.e., tiled displays or power walls, can be completely controlled using an ordinary 2D mouse. However, in case of true immersive configurations this isn't feasible anymore. Therefore, several ways of 3D user interaction are provided in VR Pack. The view can be changed using different navigation modes, a 3D menu is provided in order to control modules in 3D, and special objects like slices, selection boxes, or 3D point probes can be directly picked and manipulated using a tracked input device. In addition, a 2D mouse mode is provided allowing the user to control the conventional 2D mouse pointer using the tracked 3D mouse. In the following these features will be described in more detail.

In order to utilize 3D user interaction a tracked 3D input device with at least one button is required. By default *amira* only interprets a single button. The easiest way to make use of additional buttons is to associate *Tcl* procedures with these buttons. This is described below in a section about *Tcl event procedures*.

- *The 3D menu*
- *User-defined 3D menu items*
- *3D module controls*
- *Navigation modes*
- *Tcl event procedures*
- *The 2D mouse mode*

8.1.1 The 3D menu

The VR Pack 3D menu provides buttons allowing the user to bring up a 3D version of the user interface of every object in the Pool. In addition it can contain any number of user-defined buttons. These buttons can be configured most easily using the Tcl interface described in Section 8.2.

By default, the menu also contains a button for activating the *2D mouse mode*. This mode lets you control the 2D mouse using a 3D input device. In this way also the standard 2D user interface of *amira* can be used in a VR environment.

The basic shape of the VR Pack 3D menu is shown in Figure 8.1 on the left hand side. The menu can be manipulated in the following way:

- Initially, after activating or reloading an VR Pack configuration, the 3D menu is hidden. In order to bring up the menu double-click the button of the 3D input device. Alternatively, you may select the 3D menu toggle button of the VR Pack control module.

By default the menu will be positioned in the upper left corner of the first screen defined in the VR Pack config file. You may change the default position by setting the field *menuPosition* in the tracker section of the VR Pack *config file*. There is also a field for setting the default orientation of the menu.

- You can move the menu by pointing the wand on the blue header labeled *3D menu*, clicking the main button of the 3D input device, and then moving the device while keeping the button pressed. When the wand is on the blue header a yellow frame is displayed, indicating that this element has focus.

If you pop up the menu using the 3D menu toggle, the menu position will be reset to its default. In contrast, if you pop up the menu by double-clicking the 3D mouse button, the menu will be shown at its current position.

- In order to hide the menu again, click the red *close button* at the right side of the blue header element. Double-clicking the 3D input device again pops up the menu at its previous position.

Since version 3.1 of VR Pack the 3D menu will also be available as an ordinary 2D sub-menu under the VR menu of the *amira* main window. The 2D menu has the same structure as the 3D menu, and choosing an item from the 2D menu triggers the same actions as choosing an item from the 3D menu. This feature allows you to prepare and test a 3D menu on an ordinary desktop computer without an attached tracking system.

8.1.2 Tracker Emulator

The *Tracker Emulator* provides emulation of a 3D positioning device with two spatial sensors and two buttons. To activate it, type the word *test* into the tracker port of the VR Pack control module and press the *connect* button. After that the user interface dialog of the tracker emulator appears.

As mentioned before, the emulator emulates two spatial sensors designated as *sensor0* and *sensor1*. For each of these sensors the position is changeable by three sliders, one for each direction. Alternatively,

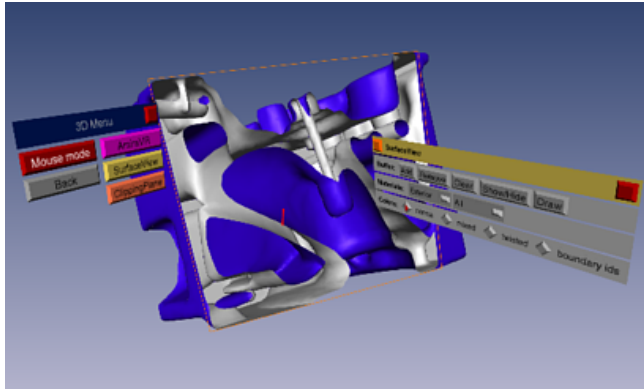


Figure 8.1: VR Pack scene with 3D menu on the left and 3D user interface of the *SurfaceView* module on the right.

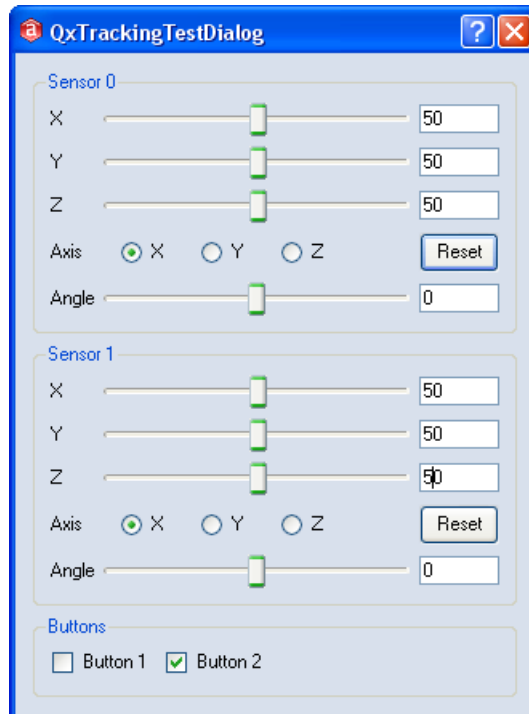


Figure 8.2: The tracker emulators user interface dialog.

as well as to reach positions of greater distance than the sliders allow, one can type the values directly into the text entry fields to the right of the slider.

The sensor's rotation can be changed by selecting one of the main axes as the rotation axis and then further adjusting the rotation angle around this axis using the slider or the entry field. To perform an additional rotation, simply select a different rotation axis and angle. The rotations are performed cumulatively. The current rotation doesn't affect the orientation of the main rotation axes. The rotation center is always the current sensor position. To set the sensor's rotation to the initial unrotated state, press the *Reset* button.

In the buttons section one can toggle the emulated button's state. As long as the checkbox is checked, the emulated button is "down". If it's unchecked, the emulated button is "up".

8.2 User-defined 3D menu items

Besides the default buttons, any number of user-defined buttons can be added to the VR Pack 3D menu using a Tcl script interface. This is useful for executing certain demos or for invoking other special actions. The script interface essentially consists of a single global Tcl method called *menu* which is provided by VR Pack. Calling the *menu* method with special parameters allows the user to add individual buttons or submenus to the main 3D menu. Each button or submenu has an id. This id can be used to modify the particular element afterwards. Whenever a button is pressed, a user-defined Tcl procedure is invoked.

An example of how to create a two-level menu hierarchy is shown below. This example is taken from the file `$AMIRA_ROOT/share/demo/tracking/demomenu.hx` contained in the VR Pack demo section. You can execute that script in order to see how the user-defined buttons work.

```
menu reset

menu insertMenu -id 0 -text "Medical"
menu insertMenu -id 1 -text "Flow Dynamics"
menu insertMenu -id 2 -text "Reconstruction"
menu insertMenu -id 3 -text "Multi-Channel"

menu 0 insertItem -id 0 -text "CT Slices" -proc "Menu0"
menu 0 insertItem -id 1 -text "Isosurface" -proc "Menu0"
menu 0 insertItem -id 2 -text "Surface model" -proc "Menu0"
menu 0 insertItem -id 4 -text "Oblique slice" -proc "Menu0"
menu 0 insertItem -id 5 -text "Pseudo-color" -proc "Menu0"

menu 1 insertItem -id 0 -text "Wing" -proc "Menu1"
menu 1 insertItem -id 1 -text "Turbine ISL" -proc "Menu1"
menu 1 insertItem -id 2 -text "Turbine LIC" -proc "Menu1"

menu 2 insertItem -id 0 -text "Slice & Isosurface" -proc "Menu2"
```

```

menu 2 insertItem -id 1 -text "Volume Rendering" -proc "Menu2"
menu 2 insertItem -id 2 -text "Simplified Surface" -proc "Menu2"

menu 3 insertItem -id 0 -text "Projection view" -proc "Menu3"
menu 3 insertItem -id 1 -text "Slicing" -proc "Menu3"
menu 3 insertItem -id 2 -text "Isosurface" -proc "Menu3"
menu 3 insertItem -id 3 -text "Volume Rendering" -proc "Menu3"

proc Menu0 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/medical/ctstack.hx } \
        1 { load $AMIRA_ROOT/share/demo/medical/isosurface.hx } \
        2 { load $AMIRA_ROOT/share/demo/medical/surf.hx } \
        3 { load $AMIRA_ROOT/share/demo/medical/tetra.hx } \
        4 { load $AMIRA_ROOT/share/demo/medical/gridcut.hx } \
        5 { load $AMIRA_ROOT/share/demo/medical/pseudocolor.hx } \
        6 { load $AMIRA_ROOT/share/demo/medical/splats.hx }
}

proc Menu1 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/cfd/wing.hx } \
        1 { load $AMIRA_ROOT/share/demo/cfd/turbine-isl.hx } \
        2 { load $AMIRA_ROOT/share/demo/cfd/turbine-lic.hx }
}

proc Menu2 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/recon/recon01.hx } \
        1 { load $AMIRA_ROOT/share/demo/recon/recon05.hx } \
        2 { load $AMIRA_ROOT/share/demo/recon/recon04.hx }
}

proc Menu3 { id } {
    global AMIRA_ROOT
    switch $id \
        0 { load $AMIRA_ROOT/share/demo/multichannel/projectionview.hx } \
        1 { load $AMIRA_ROOT/share/demo/multichannel/slicing.hx } \
        2 { load $AMIRA_ROOT/share/demo/multichannel/isosurfaces.hx } \
        3 { load $AMIRA_ROOT/share/demo/multichannel/voltex.hx }
}

```

The example above give a general idea of how to create a user-defined button hierarchy. The *menu*

command provides some additional parameters. The general form of the *menu* command is as follows:

```
menu [submenu-id [...]] cmd [options]
```

Here *submenu-id* is the id of any submenu. The root level has no id at all. The first level has one id, the second level has two ids, and so on.

Note: In contrast to previous versions of **VR Pack** now all user-defined buttons are inserted at the main level of the menu. By default, at this level there are already three buttons defined:

- a button for activating the 2D mouse mode (id 1000)
- a button for getting a list of all modules (id 1001)
- a button for getting a list of all data objects (id 1002)

You can remove these default buttons using the command `menu clear`. In order to clear the menu and then reset the default buttons, use `menu reset`. The complete list of commands is this:

`menu reset`

Removes all buttons and sub-menus from the menu and restores the default layout with entries for mouse mode, modules, and data objects. This command can only be applied at the main level.

`menu [...] clear`

Removes all buttons and sub-menus from the specified menu. If applied at the main level also the default buttons (mouse mode, modules, and data) will be removed.

`menu [...] count`

Returns the number of entries (action buttons and sub-menu buttons) in the specified menu.

`menu [...] idAt index` Returns the id of the item at position *index*.

`menu [...] insertItem [options]`

This command creates a new button and inserts it into the specified menu. The following options are available:

`-id id`

Specifies the id of the new button.

`-index index`

Specifies the position of the new button. If *index* is -1 (which is the default) the new button will be appended at the bottom of the menu.

`-text text`

Specifies the text to be displayed by the new button.

`-fg "r g b"`

Specifies the text or foreground color of the new button.

-bg "*r g b*"

Specifies the background color of the new button.

-proc *proc*

The name of the Tcl procedure to be called when the button is pressed. Either an own procedure without arguments can be used for a button. Alternatively, a procedure with one argument can be used for all buttons in a menu. The argument then is set to the id of the pressed button (see example above).

menu [...] insertMenu [options]

This command creates a new sub-menu button and inserts it into the specified menu. The id of the new button can be used as a sub-menu id for the *menu* command itself. The same options as for *insertItem* can be used. In addition the following option is available:

-type *classtype*

If this option is specified a special sub-menu will be inserted which lists all objects of type *classtype* in the *amira* Pool. For example, in order to get a list of all modules (like in the default menu) you should use insertMenu with *-type HxModule*.

menu [...] changeItem id -text *text*

Changes the text of an existing button.

menu [...] removeItem id

Removes an action button or sub-menu button from the menu.

menu [...] connectItem id *proc*

Connects a button to a Tcl procedure. The procedure will be called when the button is pressed.

menu [...] disconnectItem id *proc*

Disconnects a button from a Tcl procedure.

8.2.1 3D module controls

The 3D menu provides a button called *Modules*. Clicking on this button brings up a list of all visible modules which currently exist in the *amira* Pool. Clicking on a module button brings up a 3D version of the modules user interface. This 3D user interface looks very similar to the 2D user interface of the module which is normally shown in the Properties Area of the *amira* main window. An example of the 3D user interface of the *SurfaceView* module is shown in Figure 8.1.

The 3D user interface of every module can be moved independently from each other like the main 3D menu itself. Again, this is done by picking the header bar of the module's GUI. The GUI can be removed by clicking on the red *close button* at the right side of the header bar. On the left side an orange *viewer toggle* is displayed. As in the 2D interface a display module can be switched on or off by toggling this button.

In 3D the ports of a module can be manipulated almost in the same way as in 2D. The only remarkable difference is that text input is not possible in 3D. Text fields with numerical values like the range of a colormap port or the data window of an OrthoSlice module still can be changed using a virtual slider. This is done by clicking into the text field with the wand, and then moving the wand upwards or downwards while keeping the button pressed. As a general rule, every active element shows a yellow frame when the wand is pointing on it, i.e., when it has input focus.

8.2.2 Navigation modes

VR Pack supports two different navigation modes. The default mode is *scene manipulation*, i.e., the whole scene can be rotated and translated using the 3D input device. This is done by clicking with the wand into some empty or insensitive region in space, and then moving the wand while keeping the button pressed. In this mode the whole scene remains stick relative to the input device.

The second mode is *fly mode*. This mode can be activated by selecting the second entry from the *Mode* port of the VR Pack control module (either in the 2D or in the 3D user interface). In fly mode you can click the 3D input device at some insensitive point, and then move it in any direction. The vector from the point where the wand was clicked to the current point defines a velocity vector which is used to modify the position of the camera. The longer the vector, i.e., the more the wand is shifted, the faster the camera is moving. In the same way the orientation of the camera is modified by rotating the 3D wand. An incremental rotational change is computed by comparing the current orientation which the orientation at the point where the wand was clicked initially.

8.2.3 Tcl event procedures

By default, VR Pack treats all buttons of a 3D mouse in the same way. This allows you to operate the program even with a one-button mouse. In order to make better use of a 3D input device with multiple buttons, you can define special Tcl procedures which are called when a button is pressed or released. If these procedures return the value 1, this indicates that the event has been handled by the Tcl procedure and that it should not be further processed by *amira*. In particular, it then will not be send to the Open Inventor scene graph. The following procedures can be defined:

- `vrButton0Press`, `vrButton1Press`, ...
These procedures are called when the button with the specified index is pressed. By redefining `vrButton0Press` you can overwrite the default interaction routine.
- `vrButton0Release`, `vrButton1Release`, ...
These procedures are called when the button with the specified index is released.
- `vrButton0Db1Click`, `vrButton1Db1Click`, ...
These procedures are called when the button with the specified index is double-clicked. Note that for the first click of a double-click event an ordinary button press event is generated.

When a 3D mouse button event occurs and no appropriate Tcl event procedure is defined or if this procedure returns 0, the event will be passed to the current VR Pack event handler. The default event

handler checks if the 3D menu or some other 3D widget has been clicked. If not, it sends the event to the Open Inventor scene graph. Besides this default event handler there are also other event handlers, namely handlers for managing the different navigation modes (*examine* and *fly*).

If a 3D mouse button event was not handled by the current event handler, finally the following Tcl procedures will be called, provided they are defined:

- `vrButton0PressFallback`, `vrButton1PressFallback`, ...
- `vrButton0ReleaseFallback`, `vrButton1ReleaseFallback`, ...
- `vrButton0DbClickFallback`, `vrButton1DbClickFallback`, ...

Assuming that the default event handler is active (the one which checks the 3D menu and the Open Inventor scene graph), you can trigger certain actions when the user clicks into empty space. By default, when this happens one of the navigation mode handlers is activated. If you want to disable this feature you can define a dummy `vrButton0PressFallback` procedure which always returns 1.

8.2.4 The 2D mouse mode

The 2D mouse mode is useful for accessing GUI elements which have no direct analogon in 3D. In this mode the position of the ordinary 2D mouse pointer is controlled using the 3D input device.

- In order to activate 2D mouse mode press the red *mouse mode* button of the 3D menu. Once mouse mode is activated the **amira** main window is popped up automatically.
- You can control the 2D mouse by translating or rotating the 3D input device. Usually, rotating it is more convenient since less movement is required. The position of the 2D mouse is computed by determining the intersection of the virtual wand vector with the screen.
- In order to exit 2D mouse mode, double-click the 3D input device over some insensitive region of the 2D user interface. When leaving 2D mouse mode the main graphics window is raised automatically.

If the 2D mouse stays in the middle of the screen, of course 3D impression is disturbed. Therefore, usually it is a good idea to move the 2D mouse in a corner before returning to 3D mode.

The two Tcl procedures *AmiraVR_StartMouseMode* and *AmiraVR_StopMouseMode* are called when entering and exiting 2D mouse mode. You can use these methods for example to pop up the **amira** help browser with a page from which additional demos can be launched. For this, the following definition could be included for example in the file `$AMIRA_ROOT/share/resources/Amira.init`:

```
proc AmiraVR_StartMouseMode { } {
    global AMIRA_ROOT
    load $AMIRA_ROOT/share/usersguide/tracking.html
}
```

In order to activate or deactivate the 2D mouse mode from Tcl, the VR Pack control module provides the command `AmiraVR enableMouseMode <value>`, where `<value>` can be either 0 or 1. In order to check if 2D mouse mode is currently active, you can use `AmiraVR isMouseModeEnabled`.

8.3 Writing VR Pack custom modules

`amira` can be easily extended by writing new I/O routines, data types, modules, and other components. Details about this are described in the *Developer Pack User's Guide*.

In order to write custom modules which provide specific interaction features in a VR environment, Open Inventor nodes interpreting events generated by the 3D input device have to be inserted into the scene graph. In particular, the 3D input device generates events of type *SoTrackerEvent* and *SoControllerButtonEvent*. These two event classes have been introduced in Open Inventor 3.0. For more information about these classes please refer to the Open Inventor documentation. `amira` itself provides an additional class *Hx3DWandBase* which provides some additional information about the virtual 3D wand. Among others, this class also allows to user to highlight the wand in order to provide some visual feedback during interaction.

Below we present the source code of a sample VR Pack module which just displays a number of cubes. The cubes then can be picked and transformed using the 3D wand. The source code of the module is contained in the *Developer Pack* demo package. The particular files are called *MyVRDemo.h* and *MyVRDemo.cpp*. The module can also be directly created from the popup menu of the VR Pack control module.

Here is the listing of the header file:

```

////////////////////////////////////
//
// Illustrates 3D interaction in a VR environment
//
////////////////////////////////////
#ifndef MY_VR_DEMO_H
#define MY_VR_DEMO_H

#include <Inventor/nodes/SoSeparator.h>

#include <McHandle.h>
#include <Amira/HxModule.h>
#include <Amira/HxPortButtonList.h>
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyVRDemo : public HxModule
{
    HX_HEADER(MyVRDemo);

public:
    MyVRDemo();

```

```

        virtual void compute();
        HxPortButtonList portAction;

protected:
    ~MyVRDemo();

    McHandle<SoSeparator> scene;
    McHandle<SoSeparator> activeCube;

    bool isMoving;
    SbVec3f refPos;
    SbMatrix refMatrix;
    SbRotation refRotInverse;

    void createScene(SoSeparator* scene);
    SoSeparator* checkCube(const SbVec3f& pos);
    void trackerEvent(SoEventCallback* node);
    void controllerEvent(SoEventCallback* node);

    static void trackerEventCB(void* userData, SoEventCallback* node);
    static void controllerEventCB(void* userData, SoEventCallback* node);
};

#endif

```

Here is the listing of the source file:

```

////////////////////////////////////
//
// Illustrates 3D interaction in a VR environment
//
////////////////////////////////////
#include <stdlib.h>

#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/nodes/SoMatrixTransform.h>
#include <Inventor/events/SoTrackerEvent.h>
#include <Inventor/events/SoControllerButtonEvent.h>

#include <Amira/Hx3DWandBase.h>
#include <mypackage/MyVRDemo.h>

HX_INIT_CLASS(MyVRDemo, HxModule)

MyVRDemo::MyVRDemo() :
    HxModule(HxObject::getClassTypeId()),
    portAction(this, "action", 1)
{
    isMoving = 0;
    portAction.setLabel(0, "Reset");
    scene = new SoSeparator;

```

```

        createScene(scene);
        showGeom(scene);
    }

MyVRDemo::~MyVRDemo()
{
    hideGeom(scene);
}

void MyVRDemo::compute()
{
    if (portAction.isNew() && portAction.getIndex()==0)
        createScene(scene);
}

void MyVRDemo::createScene(SoSeparator* scene)
{
    scene->removeAllChildren();

    SoEventCallback* cb = new SoEventCallback;
    cb->addEventCallback(SoTrackerEvent::getClassTypeId(),
        trackerEventCB, this);
    cb->addEventCallback(SoControllerButtonEvent::getClassTypeId(),
        controllerEventCB, this);
    scene->addChild(cb);

    for (int j=0; j<4; j++) {
        for (int i=0; i<4; i++) {
            SoSeparator* child = new SoSeparator;

            SoMatrixTransform* xform = new SoMatrixTransform;
            SbMatrix M;
            M.setTranslate(SbVec3f(i*3.f, j*3.f, 0));
            xform->matrix.setValue(M);

            SoMaterial* mat = new SoMaterial;
            float hue = (rand()%1000)/1000.;
            mat->diffuseColor.setHSVValue(hue, 1.f, .8f);

            SoCube* cube = new SoCube;

            child->addChild(xform);
            child->addChild(mat);
            child->addChild(cube);

            scene->addChild(child);
        }
    }

    SoSeparator* MyVRDemo::checkCube(const SbVec3f& p)
    {
        for (int iChild=1; iChild<scene->getNumChildren(); iChild++) {

```

```

SoSeparator* child = (SoSeparator*) scene->getChild(iChild);
SoMatrixTransform* xform = (SoMatrixTransform*) child->getChild(0);
const SbMatrix& M = xform->matrix.getValue();

SbVec3f q;
M.inverse().multVecMatrix(p,q);
if (fabs(q[0])<1 && fabs(q[1])<1 && fabs(q[2])<1) {
    if (activeCube.ptr()!=child) {
        if (activeCube) {
            SoMaterial* mat = (SoMaterial*) activeCube->getChild(1);
            mat->emissiveColor = SbColor(0.f,0.f,0.f);
        }
        SoMaterial* mat = (SoMaterial*) child->getChild(1);
        mat->emissiveColor = mat->diffuseColor[0];
        activeCube = child;
    }
    return activeCube;
}

}

if (activeCube) {
    SoMaterial* mat = (SoMaterial*) activeCube->getChild(1);
    mat->emissiveColor = SbColor(0.f,0.f,0.f);
    activeCube = 0;
}

return 0;
}

void MyVRDemo::trackerEventCB(void* userData, SoEventCallback* node)
{
    MyVRDemo* me = (MyVRDemo*) userData;
    me->trackerEvent(node);
}

void MyVRDemo::trackerEvent(SoEventCallback* node)
{
    SoTrackerEvent* e = (SoTrackerEvent*) node->getEvent();
    Hx3DWandBase* wand = GET_WAND(e);

    if (activeCube && isMoving) {
        if (!wand->getButton(0)) {
            node->setHandled();
            isMoving = 0;
            return;
        }
    }

    SbMatrix T1; T1.setTranslate(-refPos);
    SbMatrix R; R.setRotate(refRotInverse*wand->orientation());
    SbMatrix T2; T2.setTranslate(wand->origin());

    SoMatrixTransform* xform = (SoMatrixTransform*) activeCube->getChild(0);
    xform->matrix = SbMatrix(refMatrix*T1*R*T2);

```

```

        wand->setHighlight(true);
        node->setHandled();
        return;
    }

    if (checkCube(wand->top()))
        node->setHandled();
}

void MyVRDemo::controllerEventCB(void* userData, SoEventCallback* node)
{
    MyVRDemo* me = (MyVRDemo*) userData;
    me->controllerEvent(node);
}

void MyVRDemo::controllerEvent(SoEventCallback* node)
{
    if (activeCube) {
        SoTrackerEvent* e = (SoTrackerEvent*) node->getEvent();
        Hx3DWandBase* wand = GET_WAND(e);

        if (wand->wasButtonPressed(0)) {
            SoMatrixTransform* xform = (SoMatrixTransform*) activeCube->getChild(0);
            refRotInverse = wand->orientation().inverse();
            refPos = wand->origin();
            refMatrix = xform->matrix.getValue();

            wand->setHighlight(true);
            isMoving = 1;
        }

        if (wand->wasButtonReleased(0))
            isMoving = 0;

        node->setHandled();
    }
}

```


Part IV

Large Data Pack User's Guide

Chapter 9

Large Data Pack User's Guide

9.1 Working with out-of-core data files (LDA)

The out-of-core management tools allow you load and visualize data sets larger than the amount of RAM installed on your system, as well as convert these data sets into LDA (Large Data Access) files. These LDA files can be used to visualize very large data (hundreds of gigabytes), such as seismic or microscopy data, using a limited amount of memory. It is possible to convert original data of the following types: AmiraMesh, RawData, and StackedSlices (stacks of SGI, TIFF, GIF, JPEG, BMP, PNG, JPEG2000, PGX, PNM, and RAS raster files). LDA data allows subvolume extraction to display parts of the volume, or multi-resolution access to have a full subsampled view or accurate refined local views.

In particular, the following topics will be discussed in this tutorial:

1. Adjusting the size threshold to allow conversion
2. Loading the out-of-core data
3. Raw data parameters
4. Out-of-core conversion
5. Displaying an ortho slice, an oblique slice, and a 3D volume

Please follow the instructions below. Each step builds on the step before.

9.1.1 Tune the size threshold to allow conversion

In the *Edit* menu, select the *Preferences* item. This opens the *amiraPreferences* dialog. Please select the *LDA* tab (see Figure 9.1). Using the slider or text field, set the threshold to 32MB. When you load a file of file size greater than this threshold, the out-of-core data dialog will be displayed.

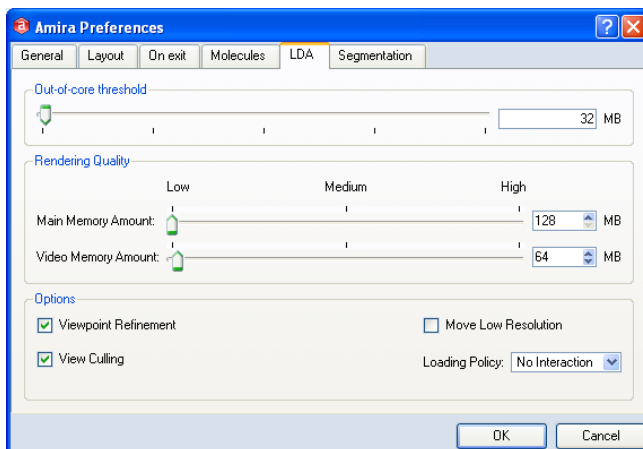


Figure 9.1: amira Preferences, LDA settings

Note: To see the images as laid out in this tutorial, you should also use the *Layout* tab of the *Edit/Preferences* menu, and toggle on *show viewer in top-level window*.

9.1.2 Load the out-of-core data

Please open the file 3DHEAD.raw using the *File/Open Data...* menu (see Figure 9.2). The file is located in data/tutorials/outofcore in the amira install directory. Its size is slightly larger than 32MB, right above the defined threshold.

The Out-Of-Core data dialog is displayed. Three loading options are displayed (see Figure 9.3):

- *Convert to LDA*: convert the file to an LDA file, and then load it.
 - *PRO*: This builds a multiresolution file allowing full interactive view or local full resolution viewing.
 - *CON*: This can be time consuming, with an initial pass and then the true conversion pass.
- *Read from disk*: read data blocks from disk, allowing almost continuous disk access.
 - *PRO*: No need to generate an extra file.
 - *CON*: Continuous access to disk. Slow with datasets larger than 4GB.
- *Read in memory*: load full data into memory and then access to memory only.
 - *PRO*: Adapted for average sized data.
 - *CON*: Requires as much RAM as your dataset size. Usually not applicable for datasets greater than 500MB or 1GB.

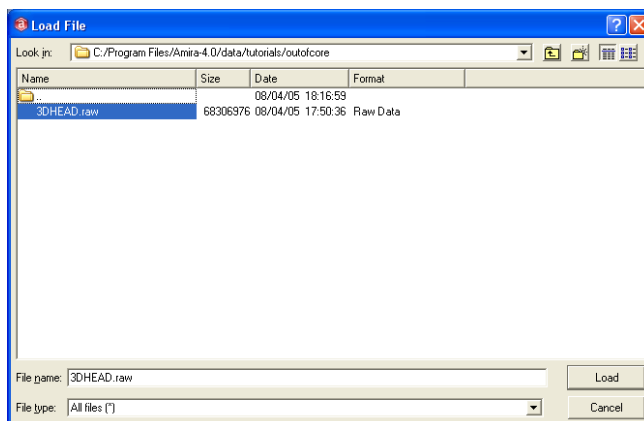


Figure 9.2: Open the out-of-core data file

Please select "Convert to LDA". Then, on the next dialog (Destination file), specify the LDA destination file. 3DHEAD.lda for instance (see Figure 9.4).

Note: An .lda file can be loaded then, without any conversion required.

Another option allows you to perform conversion in batch mode so you can run other processes while the conversion is done in the background.

9.1.3 Raw data parameters

As the input data is raw, please fill in the raw data parameters dialog with information as on the following figure (see Figure 9.5):

Data type is byte, dimension 431*431*184.

9.1.4 Out-of-core conversion

During conversion, the out-of-core conversion progress dialog is displayed (see Figure 9.6). This process is done in two steps. First of all, an initial step, and then the conversion step at about 4MB/s (on a P4 2.6GHz, no SATA disk). You can cancel the process if you wish.

The converted file is now in the Pool ready to be used and connected to other modules.

9.1.5 Display an ortho slice, an oblique slice, and a 3D volume

Right-click on the data icon in the Pool. In the Display submenu choose the OrthoSlice module. Repeat these steps for an ObliqueSlice and a Voltex (3D volume). You can also display the bounding box of

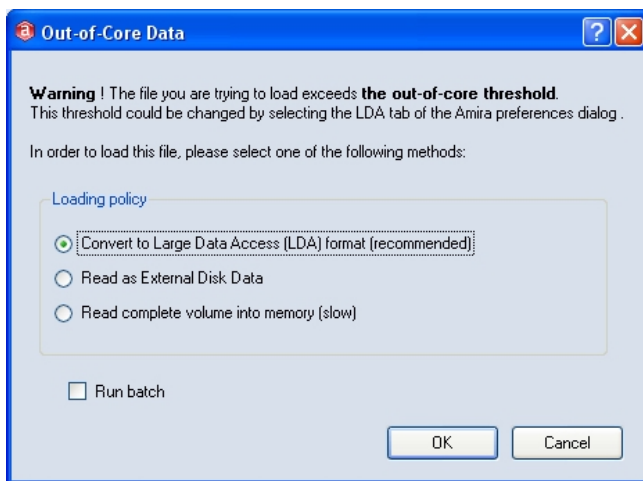


Figure 9.3: Out-of-Core data dialog

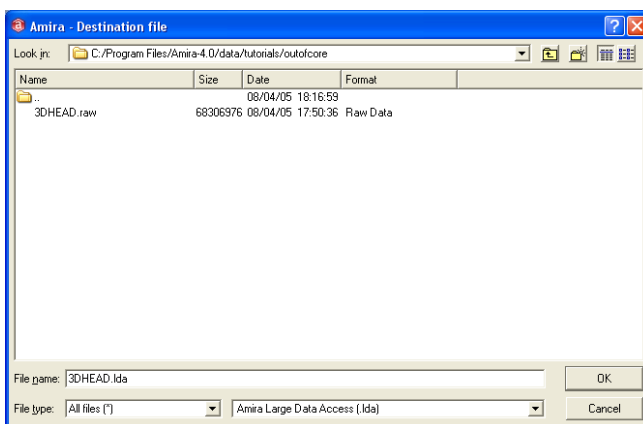


Figure 9.4: Choose LDA destination file

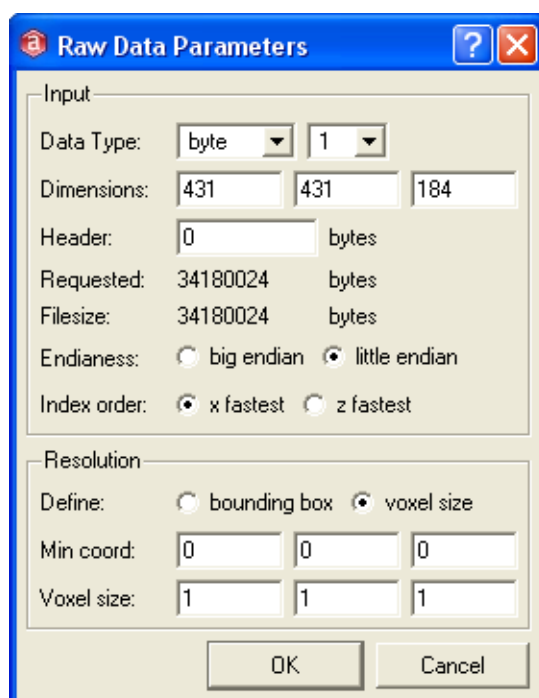


Figure 9.5: Raw data parameters panel

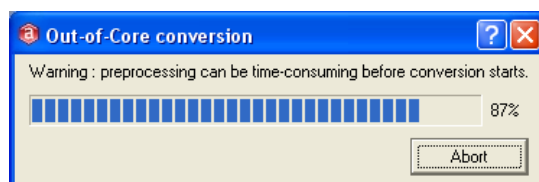


Figure 9.6: Out-of-core conversion progress dialog

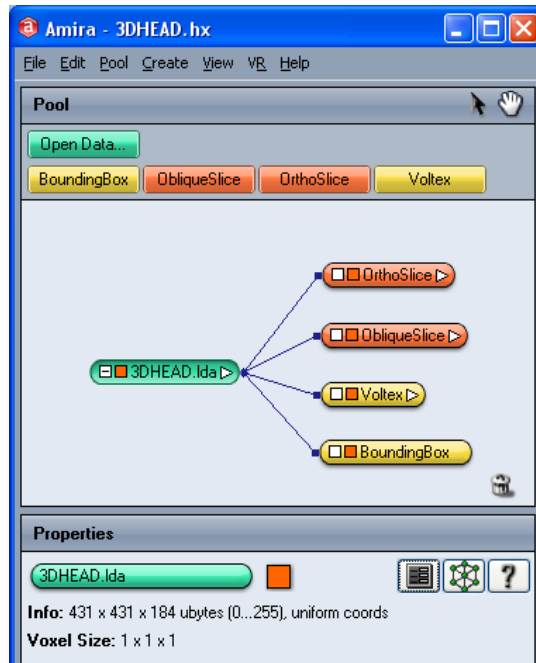


Figure 9.7: Display modules added

the full volume.

In order to view this scene with the same settings, after converting 3DHEAD.raw into 3DHEAD.Ida (Ida file required, with the right name) please load the network 3DHEAD.hx (in the same directory data/tutorials/outofcore).

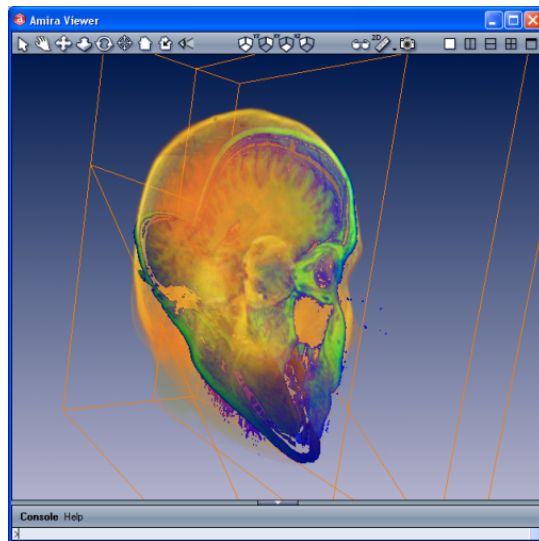


Figure 9.8: Network display

Part V

Quantification+ Pack User's Guide

This extension integrates the Visilog product by Noesis into **amira**. The Quantification+ Pack, with its the wide range of image processing tools, can be used together with the 3D visualization and geometry reconstruction capabilities of **amira**.

A set of demos is provided. For details, see the documentation of the *Visilog* module.

Index

- .Amira, [114](#), [142](#)
- affine transformations, [109](#)
- amira
 - class structure, [106](#)
 - data objects, [3](#)
 - modules, [3](#)
 - packs, [7](#)
- amira Developer Pack, [7](#)
- amira DICOM Reader, [7](#)
- amira Large Data Pack, [7](#)
- amira Mesh Pack, [7](#)
- amira Molecular Pack, [7](#)
- amira Quantification+ Pack, [7](#)
- amira SEG-Y Reader, [7](#)
- amira Very Large Data Pack, [7](#)
- amira VR Pack, [7](#)
- Amira.init, [114](#), [142](#)
- AMIRA_LOCAL, [114](#), [129](#)
- AMIRA_ROOT, [129](#)
- auto-save, [101](#)
- auto-select modules, [101](#)
- camera trackball, [92](#)
- CATIA 4, [8](#)
- CATIA 5, [8](#)
- color depth, [116](#)
- command line options, [112](#)
- compute indicator, [101](#)
- coordinates, [108](#)
- Create menu, [83](#)
- data import, [111](#)
- database
 - default, [81](#)
 - user-defined, [81](#)
- default directories, [113](#)
- Edit menu
 - Copy, [80](#)
 - Cut, [80](#)
 - Database, [81](#)
 - Delete, [80](#)
 - Paste, [80](#)
 - Preferences, [81](#)
 - Select All, [81](#)
- editors, [3](#)
- environment variables, [113](#)
- F1 key, [129](#)
- features, [4](#)
- file dialog
 - changing directories, [98](#)
 - filename filter, [98](#)
 - popup menu, [98](#)
 - selecting files, [98](#)
- File menu
 - Jobs, [80](#)
 - Open data, [77](#)
 - Open Time Series, [78](#)
 - Quit, [80](#)
 - Recent Files, [79](#)
 - Recent Networks, [80](#)
 - Save Data, [78](#)
 - Save Data As, [78](#)
 - Save Network, [79](#)
- firing algorithm, [100](#)
- font size, [114](#), [115](#)

- function key, [115](#)
 - procedure, [142](#)
- help
 - for commands, [129](#)
 - help browser, [85](#)
 - searching, [86](#)
- hidden data objects, [101](#)
- hot-key procedure, [115](#), [142](#)
- IGES, [8](#)
- Job dialog box, [99](#)
- Mecalog Radioss For amira, [8](#)
- OpenGL driver, [116](#)
- parameters of data objects, [110](#)
- Pool, [88](#)
- Pool menu
 - Duplicate, [82](#)
 - Hide, [81](#)
 - Remove, [81](#)
 - Remove All, [82](#)
 - Rename, [82](#)
 - Show, [82](#)
 - Show All, [82](#)
- Port, [88](#)
- preferences, [100](#)
- regular grid, [108](#)
- ResolveRT Pack, [8](#)
- save network, [101](#), [141](#)
- scalar fields, [107](#)
- SCRIPTDIR, [129](#)
- SCRIPTFILE, [129](#)
- Scripting interface, [121](#)
- Skeleton Pack, [8](#)
- Snapshot dialog box, [104](#)
- Spacemouse, [114](#)
- start-up script, [114](#), [142](#)
- STEP, [8](#)
- stereo mode, [114](#)
- surface, [109](#)
- swap space, [116](#)
- system information dialog, [105](#)
- system requirements, [115](#)
 - HP-UX, [117](#)
 - Linux, [117](#)
 - Mac, [118](#)
 - Silicon Graphics, [116](#)
 - SunOS, [117](#)
 - Windows, [116](#)
- system stability, [116](#)
- Tcl, [121](#)
- Tcl introduction, [122](#)
- tetrahedral grids, [108](#)
- TNO Madymo For amira, [8](#)
- Tracker Emulator, [210](#)
- vector fields, [108](#)
- VertexSets, [109](#)
- View menu
 - Axis, [85](#)
 - Background, [83](#)
 - Console, [85](#)
 - Easy fade, [85](#)
 - Fading effect, [85](#)
 - Fog, [84](#)
 - FPS (frames-per-second), [85](#)
 - Frame counter, [85](#)
 - Layout, [83](#)
 - Lights, [84](#)
 - Measuring, [85](#)
 - Transparency, [83](#)
- Viewer, [92](#)
 - camera trackball, [92](#)
 - Fullscreen, [95](#)
 - Home, [93](#)
 - Interact, [92](#)
 - interaction mode, [92](#)
 - Layout, [95](#)
 - Measuring, [94](#)
 - Perspective/Ortho toggle, [93](#)

- Pick, [92](#)
- rotate button , [93](#)
- Seek, [93](#)
- Set Home, [93](#)
- Snapshot, [94](#)
- Stereo, [94](#)
- Trackball, [92](#)
- Translate, [93](#)
- View, [92](#)
- View All, [94](#)
- viewing directions Axial, Coronal, Sagittal, [94](#)
- viewing directions YZ, XZ, XY, [94](#)
- viewing mode, [92](#)
- Zoom, [93](#)
- zoom, [92](#)
- viewer toggles, [101](#)
- virtual memory, [116](#)
- work area, [91](#)